Aspektorientierte Datenhaltung in Produktkonfiguratoren – Anforderungen, Konzepte und Realisierung

Dissertation

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

vorgelegt dem Rat der Fakultät für Mathematik und Informatik der
Friedrich-Schiller-Universität Jena

von
Dipl.-Inf. Matthias Liebisch
geboren am 2. März 1978 in Jena

Gutachter:

1. Prof. Dr. Klaus Küspert, Friedrich-Schiller-Universität Jena
2. Prof. Dr. Nils Boysen, Friedrich-Schiller-Universität Jena
3. Dr. Georg Elsner, ORISA Software GmbH

Tag der Einreichung: 10.01.2014
Tag der öffentlichen Verteidigung: 03.04.2014

Meiner Familie

Danksagung

An dieser Stelle möchte ich denjenigen Menschen danken, die mich auf dem Weg zur vorliegenden Dissertationsschrift begleitet und unterstützt haben.

Besonderer Dank gilt zunächst meinem Doktorvater Herrn Prof. Dr. Klaus Küspert. Erst durch sein persönliches Engagement wurde mir die Möglichkeit eröffnet, als wissenschaftlicher Mitarbeiter an seinem Lehrstuhl in die Welt von Forschung und Lehre einzutauchen. In jenen fünf Jahren sorgten etliche fachliche sowie außerfachliche Gespräche für ein durchweg angenehmes und lockeres Arbeitsklima. Darüber hinaus gebühren ihm Respekt und Anerkennung für die langjährige Organisation vieler zusätzlicher Veranstaltungen wie Produkt-Zertifizierungen, Praxis-Exkursionen und Lehrstuhl-Datenbanktage. Ich bin glücklich und dankbar, dass diese einzigartigen Gelegenheiten sowohl mein Studium als auch die Zeit danach enorm bereichert haben.

Ebenso danke ich ganz herzlich Herrn Dr. Georg Elsner als Geschäftsführer der ORISA Software GmbH für seine Unterstützung. Dies betrifft vor allem die Förderung meiner beruflichen und sozialen Weiterentwicklung seit meinem Einstieg im Unternehmen. Ein auf Teilzeit reduziertes Arbeitsverhältnis während der fünf Jahre als Doktorand an der FSU Jena stellt diesbezüglich den bisherigen Höhepunkt dar. Weiterhin gilt ihm mein ausdrücklicher Dank für anregende Diskussionen grundlegender Ideen zur inhaltlichen Fokussierung der Dissertation sowie für die Übernahme des externen Gutachtens.

Auch Herrn Prof. Dr. Nils Boysen von der FSU Jena danke ich für sein Interesse am Dissertationsthema und die Übernahme des Zweitgutachtens.

Mein Dank geht weiterhin an die zahlreichen Studenten, deren gute und zum Teil hervorragende Studien-, Bachelor- und Diplomarbeiten wesentlich zum Gelingen der vorliegenden Arbeit beigetragen haben. In diesem Zusammenhang seien ausdrücklich die Leistungen von Andreas Göbel, Andreas Krug und Bernhard Pietsch hervorgehoben. Daneben möchte ich mich bei Carola Eichner sowie meinen ehemaligen Kollegen Dr. Klaus Friedel, Dr. Thomas Müller, Dr. Gennadi Rabinovitch, Dr. David Wiese, Andreas Göbel, Katharina Büchse und Bernhard Pietsch für das sehr entspannte Arbeitsumfeld am Lehrstuhl mit einigen erfrischenden Momenten bedanken.

Schließlich geht ein ganz lieber Dank an meine Familie und Eltern für den Rückhalt auf sozialer und emotionaler Ebene. Insbesondere danke ich meiner Frau Diana für ihre moralische Unterstützung und die aufgebrachte Toleranz, wenn so mancher gemeinsamer Abend dem Schreiben geopfert werden musste! Die dafür notwendige Ruhe und Erholung ist nicht zuletzt unserem Sohn Felix und seinen fast ausnahmslos durchgeschlafenen Nächten zu verdanken.

Jena, 10.01.2014 Matthias Liebisch

Kurzfassung

Seit Mitte des 20. Jahrhunderts äußert sich der weltweite soziokulturelle Prozess der Individualisierung auch im geänderten Konsumverhalten. Dabei sehen sich Unternehmen zunehmend mit dem Wunsch nach kundenindividuellen Produkten zum Preis eines vergleichbaren Massenprodukts konfrontiert. Während durch produktionsspezifische Konzepte wie Modularisierung, Baukasten-Prinzip oder Gleichteile-Strategie zwar eine enorme Variantenvielfalt erreicht werden kann, sind jedoch zur Beherrschung der damit verbundenen Komplexität sowie zur Sicherstellung von fachlich konsistenten Zusammenstellungen sogenannte **Produktkonfiguratoren** als unterstützende IT-Anwendungssysteme notwendig. Gleichzeitig wurde und wird deren Einsatz als unmittelbarer Kontakt zwischen Unternehmen und Konsumenten auch durch die Verbreitung des Internets ab den 1990er Jahren begünstigt. Eine detaillierte Einführung, Beschreibung und Klassifizierung von Produktkonfiguratoren ist Inhalt des ersten Teils der vorliegenden Arbeit.

Aufgrund des meist internationalen Einsatzes von Produktkonfiguratoren im webbasierten E-Commerce-Umfeld müssen sie sich auch den Anforderungen der Globalisierung stellen, welche generell für informationstechnische Systeme in diesem Kontext gelten. Dabei wirken sich die regionalen Unterschiede bezüglich Sprache, Kultur oder Preispolitik nicht nur auf die offensichtliche Darstellung der Nutzeroberfläche aus, sondern haben auch Einfluss auf die Struktur und Speicherung der Produkte im jeweils zugrunde liegenden Datenmodell. Allerdings ist insbesondere zur Gewährleistung der Datenunabhängigkeit eine generische und orthogonale Integration jener als funktionale Aspekte bezeichneten Dimensionen erstrebenswert. Zu diesem Zweck werden im zweiten Teil der Arbeit die **aspektorientierte Datenhaltung** als neuartiges kanonisches Paradigma formuliert und Möglichkeiten zur Persistierung analysiert.

Nach der bisherigen theoretischen Einführung und der Anforderungs-Definition erfolgt im dritten Teil der Entwurf eines **Referenzmodells** für die aspektorientierte Datenhaltung. Als Grundlage hierfür dient das relationale Datenmodell, das sich unter allen zuvor verglichenen Persistierungsvarianten insbesondere aufgrund seiner großen Praxisrelevanz für Datenbanksysteme qualifiziert. In diesem Zusammenhang wird die Funktionsweise zur Speicherung aspektspezifischer Daten anhand eines Anwendungsbeispiels erläutert. Darüber hinaus finden auch die Diskussion des notwendigen Zugriffsmodells sowie eine Gegenüberstellung von geeigneten Zugriffstechniken für die Auswertung jener Tabellenstrukturen im Referenzmodell statt.

Abschließend wird im letzten Teil der Arbeit die Praxistauglichkeit des beschriebenen Referenzmodells für die aspektorientierte Datenhaltung unter Beweis gestellt. Dazu erfolgt zunächst die Vorstellung zentraler Klassen und Methoden sowie relevanter Details einer **prototypischen Implementierung** der als Anwendungsbibliothek ausgelegten Zugriffsschicht. Diese ermöglicht neben dem Zugriff auf die eigentlichen aspektspezifischen Daten auch die Verwaltung der zugehörigen Metadaten im Referenzmodell. Daraufhin finden die Bewertung und der Vergleich mit alternativen Zugriffstechniken im Rahmen eines umfangreichen Performancetests statt. Im Ergebnis zeigt sich dabei, dass die Verwendung des Referenzmodells für die aspektorientierte Datenhaltung mit Hilfe der präsentierten Anwendungsbibliothek effizient möglich ist.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Dieses Kapitel motiviert zunächst in **Abschnitt 1.1** den Einsatz von Produktkonfiguratoren und die damit verbundenen Impulse zur Konzeption des Paradigmas der aspektorientierten Datenhaltung. Anschließend werden in **Abschnitt 1.2** die zur Realisierung erforderlichen Aufgaben und Ziele beschrieben, bevor im **Abschnitt 1.3** auf die Gliederung der Arbeit eingegangen wird. Hinweise zu typografischen Konventionen in **Abschnitt 1.4** runden das Kapitel ab.

1.1 Motivation

> *Das Glück besteht darin, zu leben wie alle Welt*
> *und doch wie kein anderer zu sein.* Simone de Beauvoir

Unsere heutige Gesellschaft und damit die persönliche Entwicklung jedes Einzelnen ist zu großen Teilen geprägt durch einen wachsenden Anspruch auf Selbstbestimmung und Entfaltung. Dieser als **Individualisierung** bezeichnete soziokulturelle Prozess lässt sich bereits im 16. Jahrhundert durch das verstärkte Auftreten von Selbstporträts, Autobiographien und Tagebüchern beobachten [Jun02]. Bedingt durch die spätere Phase der Industrialisierung zeigten sich weitere charakteristische Merkmale wie die Arbeitsteilung, der Zerfall von Großfamilien und die Urbanisierung. Zudem erfolgt laut [Bec83] eine Auflösung von gesellschaftlichen Zuordnungen bezüglich Stand und Klasse seit Mitte des 20. Jahrhunderts. Stattdessen übernimmt zunehmend ein verändertes Konsumverhalten die Rolle der Abgrenzung gegenüber dem sozialen Umfeld durch den Erwerb individueller und maßangefertigter Produkte und Dienstleistungen [BGHK01].

Diese Art der Nachfrage stellt Unternehmen vor ein Spannungsfeld mit neuen und teilweise sich widersprechenden Herausforderungen. Einerseits wünschen Kunden die Möglichkeit, zunehmend mehr charakteristische Eigenschaften des zu erwerbenden Produkts beeinflussen zu können. Andererseits wird dadurch aber, bis auf Ausnahmen im Sektor der Luxusgüter und Sonderanfertigungen, kein oder nur ein geringer Anstieg des Kaufpreises gegenüber dem „Produkt von der Stange“ akzeptiert. Zudem entspricht diese Differenzierung nicht der von Unternehmen üblicherweise angestrebten Homogenisierung der Nachfrage im Sinne der Kostenreduzierung [Lev83]. Die Bewältigung dieser widersprüchlichen Anforderungen zeigt sich seit den 1970er Jahren in der zunehmenden Ablösung der Massenproduktion, welche sich im Rahmen der Industrialisierung herausbildete und deren

Effektivität von Henry Ford mit Hilfe der Fließbandtechnik maximiert wurde [vGO26]. Stattdessen findet die kundenindividualisierte Massenproduktion eine stetig wachsende Anwendung [Grä04]. Für diese auch als Mass Customization [Pin93] bezeichnete Art der Fertigung haben sich sogenannte **Produktkonfiguratoren** als sinnvolle informationstechnische Unterstützung durchgesetzt. Diese gewährleisten die Zusammenstellung eines Produkts gemäß den spezifizierten Kundenanforderungen auf Basis zuvor definierter Komponenten und Regeln. Dabei wird durch geeignete Modularisierung aller relevanten Produkte die Verwendung standardisierter und damit kostengünstiger Baugruppen möglich, während gleichzeitig dem Kunden der Eindruck des individuell gefertigten Produkts vermittelt werden kann.

Neben der Individualisierung ist unter dem Schlagwort **Globalisierung** ein weiterer sehr tiefgreifender Prozess zu beobachten, dessen Wurzeln sich bis ins Spätmittelalter zu Persönlichkeiten wie Christoph Kolumbus und Jakob Fugger [Bru09] zurückverfolgen lassen. Heute mehr denn je versteht man unter Globalisierung insbesondere die politische, kulturelle sowie wirtschaftliche Verflechtung zwischen Staaten, Unternehmen und Individuen [Lev83]. Die ökonomische Bedeutung zeigt sich vor allem am Wachstum des weltweiten Außenhandelsvolumens seit Mitte des 20. Jahrhunderts gegenüber dem Wachstum der Weltwarenproduktion [HBM+11]. Voraussetzung hierfür waren einerseits die Entwicklungen im Transportbereich, beginnend mit der Einführung der Standardcontainer durch Malcom McLean im Jahr 1956, welche ein viel effizienteres Beladen von Schiffen im Vergleich zu traditionellen Gebinden ermöglichten [Cud06]. Andererseits spielen auch Fortschritte in der Kommunikationstechnologie im Rahmen der Digitalen Revolution bis zur heute selbstverständlichen Nutzung des Internets eine entscheidende Rolle [Tap96].

Für global agierende Unternehmen ergeben sich mit der Erschließung neuer Absatzmärkte zusätzliche Herausforderungen aufgrund von nationalen bzw. regionalen Differenzen im Bereich von Kultur, Sprache, Gesetzeslage oder Preisstruktur. Beispielsweise ist der Vertrieb von Autos in Großbritannien nur als Rechtslenker mit englischen Bezeichnungen und in Britischen Pfund sinnvoll. Diese Dimensionen der Differenzierung, welche in der vorliegenden Arbeit als **funktionale Aspekte** bezeichnet werden, haben Einfluss auf die Ausgestaltung aller Produkte eines Unternehmens sowie auf die Verarbeitung durch einen zur Kundenunterstützung eingesetzten Produktkonfigurator. Dabei spielen neben der Darstellung insbesondere die Abbildung, Speicherung und Verwaltung der mit funktionalen Aspekten verbundenen zusätzlichen Daten im zugrunde liegenden Datenmodell des Produktkonfigurators eine wichtige Rolle. Hierfür liefern bisher weder Forschung noch Praxis kanonische Ansätze oder nutzbare Produkte, welche das dynamische Hinzufügen oder Entfernen funktionaler Aspekte ohne tiefgreifende Modellanpassungen ermöglichen.

Die theoretische Konzeption und prototypische Realisierung des Paradigmas der **aspektorientierten Datenhaltung**, welches auf generische Weise die anwendungsabhängige Unterstützung beliebiger funktionaler Aspekte in einem bereits existierenden oder neu entworfenen Datenmodell zulässt, bildet den **Schwerpunkt dieser Arbeit**. Dabei sollen für das betreffende Datenmodell typische anzustrebende Eigenschaften wie Modularität, Redundanzfreiheit, Wartbarkeit und Wiederverwendbarkeit durch die Integration funktionaler Aspekte auch weiterhin sichergestellt werden können. Diesbezüglich findet initial eine Analyse und formale Aufarbeitung zur grundlegenden Charakteristik funktionaler Aspekte statt. Schließlich ist auch die Durchführung sowie Auswertung verschiedener Performancetests ein wichtiger Bestandteil dieser Arbeit, um die praxisrelevante Einsatzfähigkeit des Paradigmas zu prüfen.

1.2 Ziele und Aufgaben

Die wichtigsten Ziele der vorliegenden Arbeit und die daraus resultierenden Aufgaben lassen sich wie folgt zusammenfassen:

Beschreibung und Klassifikation von Produktkonfiguratoren
: Es sollen im Rahmen einer wissenschaftlichen Auseinandersetzung, soweit für die aspektorientierte Datenhaltung erforderlich, der Aufbau und die Funktionsweise von Produktkonfiguratoren erläutert werden. Dabei ist insbesondere eine detaillierte Analyse und Klassifikation der zugrunde liegenden Datenverwaltung zu erarbeiten.

Vorstellung eines Grundmodells für Produktkonfiguratoren
: Es soll durch die Entwicklung einer Referenzarchitektur ein Beitrag auf dem Weg zur Standardisierung sowie zur besseren Vergleichbarkeit bezüglich Funktionsumfang und Einsatzmöglichkeiten verschiedener heute existierender Ansätze von Produktkonfiguratoren erbracht werden.

Analyse und Formalisierung von funktionalen Aspekten
: Es sollen die charakteristischen Eigenschaften funktionaler Aspekte und deren Auswirkungen auf die Datenhaltung klassifiziert und analysiert werden. Neben der Entwicklung eines formalisierten Zugriffskonzepts für aspektspezifische Daten steht weiterhin die Betrachtung derartiger mehrdimensionaler Einflüsse in anderen informationstechnischen Bereichen wie der Softwareentwicklung im Vordergrund.

Konzeption des Paradigmas der aspektorientierten Datenhaltung
: Es soll zur Bewältigung der in Produktkonfiguratoren identifizierten Datenhaltungsherausforderungen aufgrund der Existenz funktionaler Aspekte ein Modellierungsparadigma zur kanonischen Abbildung dieser Aspekte entwickelt werden. Hierbei steht vor allem die Formulierung wünschenswerter Anforderungen wie Modularität und Orthogonalität bezüglich der Abbildung aspektspezifischer Daten im Mittelpunkt.

Analyse und Bewertung von Persistierungsmöglichkeiten
: Es sollen verschiedene Speicherkonzepte und Realisierungsalternativen zur Abbildung funktionaler Aspekte untersucht und miteinander verglichen werden. Die hierfür zum Einsatz kommenden Bewertungskriterien sind einerseits direkt aus dem Paradigma der aspektorientierten Datenhaltung abzuleiten und müssen andererseits technischen und praktischen Anforderungen, wie beispielsweise einer performanten Verarbeitungsleistung, Rechnung tragen.

Definition eines Referenzmodells
: Es soll ausgehend von der vorgelagerten Bewertung verschiedener Möglichkeiten zur Speicherung funktionaler Aspekte ein geeignetes Referenzmodell entwickelt und beschrieben werden, welches sich insbesondere im vorgestellten Grundmodell für Produktkonfiguratoren integrieren lässt. In diesem Kontext spielt auch die Konzeption einer performanten Zugriffsschicht eine entscheidende Rolle.

Nachweis der Realisierbarkeit und Praxistauglichkeit
: Es soll nachgewiesen werden, dass das in der vorliegenden Arbeit konzipierte Referenzmodell sowohl für die Speicherung von funktionalen Aspekten als auch für den Zugriff auf Daten unter Einfluss von Aspekten realisierbar ist. Die Bestätigung wird durch eine prototypische Implementierung sowie einen Performancetest erbracht.

1.3 Aufbau der Arbeit

Das **Kapitel 2** widmet sich einer Einführung und Beschreibung von Produktkonfiguratoren, wobei in einem ersten Teil die Motivation für den Einsatz dieser Technologie sowie relevante Begriffe erläutert werden sollen. Darauf aufbauend stehen im zweiten Teil die Klassifikation charakterisierender Merkmale von Produktkonfiguratoren und die Skizzierung eines Grundmodells im Vordergrund. Als abschließende Überleitung zum nächsten Kapitel werden die Herausforderungen und Auswirkungen der Globalisierung auf die Datenhaltung in Produktkonfiguratoren genauer analysiert.

In **Kapitel 3** wird das Paradigma der aspektorientierten Datenhaltung zur Modellierung und Abbildung funktionaler Aspekte vorgestellt. Dazu findet einleitend in einem theoretischen Teil die Charakterisierung und Einordnung funktionaler Aspekte und deren Auswirkungen statt, anschließend werden die konkreten Anforderungen an das Paradigma formuliert. Im zweiten, praxisnäheren Teil stehen die Analyse und Bewertung möglicher Realisierungskonzepte für die aspektorientierte Datenhaltung in verschiedenen Persistierungsmodellen wie relationale oder objektrelationale Datenbankmanagementsysteme im Fokus der Betrachtungen.

In **Kapitel 4** folgt als Konsequenz aus der vorangegangenen Bewertung verschiedener Persistierungsmodelle die Vorstellung und Beschreibung des relationalen Referenzmodells für die aspektorientierte Datenhaltung. Neben der Erläuterung von Aufbau und Wirkungsweise der Tabellenstrukturen im Modell sowie dessen ganzheitlicher Bewertung werden anschließend Alternativen für praktikable Zugriffstechniken skizziert und einer Evaluation unterzogen. Außerdem findet durch eine formalisierte Betrachtung der aspektspezifischen Daten eine zentrale Begriffsprägung statt, welche als Grundlage für die im nachfolgenden Kapitel eingeführte Zugriffsschicht dient.

In **Kapitel 5** wird ausgehend von den Ergebnissen der bisherigen Evaluationen die Verwendung einer Programmierschnittstelle als geeignete Zugriffsschicht auf das Referenzmodell motiviert. Anschließend erfolgt sowohl die Betrachtung genereller architektonischer Fragen als auch eine umfassende Beschreibung der konkreten spezifizierten Klassenstrukturen und Datenobjekte der prototypischen Implementierung. Diese Ausführungen werden abgerundet durch die Analyse und Veranschaulichung der Funktionsweise der eigentlichen Anfrageverarbeitung.

In **Kapitel 6** wird die beschriebene Programmierschnittstelle einem Performancetest unterzogen. Anhand einer Menge von Anfragen bezüglich eines realistischen Anwendungsbeispiels soll geprüft werden, ob die für einen Praxiseinsatz notwendigen Reaktionszeiten beim Zugriff auf aspektspezifische Daten mit Hilfe des Referenzmodells und der implementierten Zugriffsschicht erreichbar sind. Parallel dazu wird die standardisierte SQL-Schnittstelle ebenfalls mit dieser Anfragelast konfrontiert, um für die Bewertung aller ermittelten Messwerte eine sinnvolle Vergleichsbasis zu erhalten.

Abschließend fasst **Kapitel 7** die wichtigsten Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf angrenzende Themen.

1.4 Typografische Konventionen

Für eine bessere Lesbarkeit unterliegt der Text einigen typografischen Festlegungen, auf die hier hingewiesen werden soll. Das jeweils erste Vorkommen einer Referenz auf eine Abbildung oder einen Gliederungspunkt bis zur zweiten Ordnung ist mit Schriftdicke **fett** hervorgehoben, dies gilt auch für Schwerpunkte in der Arbeit. Wichtige Begriffe sind dagegen in *kursiver* Schrift und innerhalb ihrer Definition ***fett kursiv*** gesetzt. Die Namen von Tabellen und Attributen werden mit Hilfe von Kapitälchen markiert, während Syntax und Code-Fragmente mittels `nichtproportionaler` Typewriter-Schrift vom übrigen Text abgesetzt sind. Schließlich wird für Java-Quelltext eine serifenlose Schrift verwendet, wobei **Klassennamen** und *Methoden* zusätzlich formatiert sind.

Die Darstellung von Relationen und deren Beziehungen folgt der in **Abbildung 1.1** verwendeten Notation, wobei hier beispielhaft die Zusammenhänge zwischen Flüssen, Meeren und Ländern modelliert wurden. Die Primärschlüssel-Attribute einer Relation sind mit dem Schlüssel-Icon ¶ gekennzeichnet, für Schlüsselkandidaten steht dieses in Klammern. Weiterhin ist zu einem Attribut der Datentyp in eckigen Klammern angegeben, ein Stern (*) symbolisiert ein optionales Attribut (NULL-Wert ist erlaubt). Schließlich können zwei Relationen über Fremdschlüssel miteinander in Beziehung stehen, welcher als durchgezogene Linie mit Pfeil zwischen zwei Attributen dargestellt wird. Ist dagegen eine solche Linie gestrichelt, spezifiziert dies die möglichen Datenquellen zu einem Attribut. Für die Sicherstellung eines solchen „generalisierten Fremdschlüssels“ ist jedoch die Anwendung selbst verantwortlich, weil ein solches Konstrukt im relationalen Datenmodell nicht existiert. Notwendig ist dies beispielsweise, um ausdrücken zu können, dass ein Fluss (Mündung.ZuflussID) nicht nur in einen anderen Fluss (Mündung.ZielID = Fluss.FlussID), sondern auch in ein Meer (Mündung.ZielID = Meer.MeerID) münden kann. Dabei wurden Flüsse und Meere aufgrund ihrer Verschiedenheit bewusst als eigene Relationen modelliert.

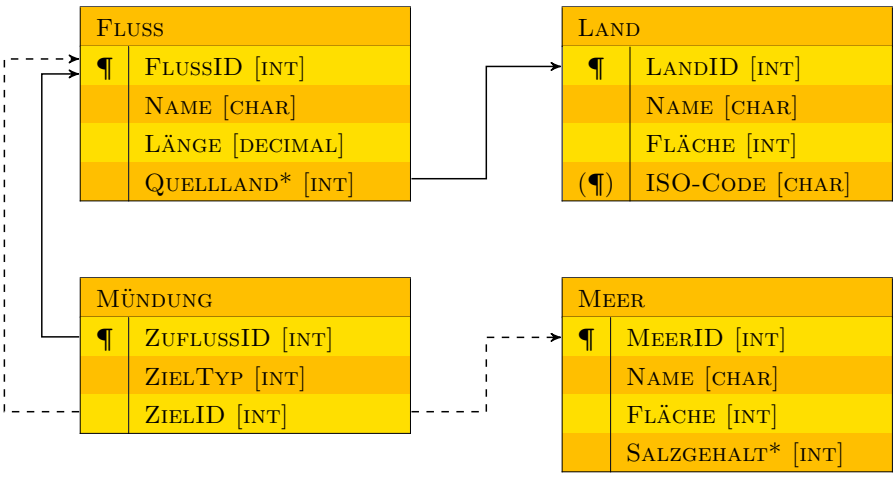


Abbildung 1.1: Beispiel für die Darstellung von Relationen

Im Literaturverzeichnis sind aufgrund der Relevanz für die Dissertationsschrift diejenigen Publikationen **hervorgehoben**, welche vom Autor dieser Arbeit selbst verfasst oder von durch ihn betreute Studenten erstellt wurden. Die Gültigkeit aller ausgewiesenen Web-Links wurde am 10.01.2014 validiert.

Kapitel 2

Produktkonfiguratoren

Aufgrund des einleitend geschilderten Wechsels von der Massenproduktion zur kundenindividualisierten Massenproduktion erfreuen sich Produktkonfiguratoren für die Bewältigung des damit einhergehenden Komplexitätsanstiegs wachsender Beliebtheit. In diesem Kapitel sollen jene Anwendungssysteme näher betrachtet werden. Beginnend mit den relevanten Grundlagen zum Produktverständnis in **Abschnitt 2.1**, erfolgt eine Definition zur Abgrenzung von Produktkonfiguratoren in **Abschnitt 2.2**, bevor deren charakteristische Merkmale in **Abschnitt 2.3** beschrieben werden. Auf dieser Basis findet der Entwurf eines integrativen Grundmodells in **Abschnitt 2.4** statt. Schließlich werden Herausforderungen für Produktkonfiguratoren im Kontext der Globalisierung in **Abschnitt 2.5** analysiert.

2.1 Grundlagen

Zur Vorbereitung der weiteren Ausführungen im vorliegenden Kapitel werden in diesem Abschnitt die wichtigsten Begrifflichkeiten und Zusammenhänge aus mikroökonomischer Sicht erläutert. Neben der Charakterisierung des abstrakten Produkts in Abschnitt 2.1.1 findet weiterhin in Abschnitt 2.1.2 eine Analyse und Bewertung der verschiedenen Prozesse zur Produktfertigung sowie deren Eignung im Rahmen der Produktkonfiguration statt.

2.1.1 Produkt

Dieser Unterabschnitt setzt sich einleitend mit dem Produkt-Terminus und der umgebenden Begriffswelt auseinander. Anschließend wird eine Klassifikation zur Abgrenzung unterschiedlicher Produktarten vorgestellt.

2.1.1.1 Definitionen

Nicht nur in der deutschen Sprache wird der Begriff *Produkt* homonym verwendet, welches in dem lateinischen Wort *producere* im Sinne von „hervorbringen“ seinen Ursprung hat. Unter anderem steht ein Produkt für das Ergebnis einer mathematischen Rechenoperation oder als Resultat einer chemischen Reaktion. Darüber hinaus existiert vor allem eine wirtschaftliche Bedeutung, auf welche im weiteren Verlauf des Abschnitts näher eingegangen werden soll.

Wie in [Kru10] aufgezeigt, gibt es innerhalb des Wirtschafts- und Wissenschaftsgebiets bezüglich des Produktbegriffs sehr unterschiedliche Sichtweisen, die entweder mehr die wirtschaftliche Verwendung oder eher den technischen Aufbau eines Produkts fokussieren. Beispielsweise wird in [KASW06] ein Produkt als Objekt angesehen, welches auf einem Markt angeboten wird und zum Kauf, zur Verwendung oder zum Verbrauch geeignet ist und damit letztlich der Bedürfnisbefriedigung des Konsumenten dient. Dagegen ist durch [Bie01] das Produkt als ein gebrauchs- beziehungsweise verkaufsfähiges Erzeugnis von materieller oder immaterieller Natur definiert, welches unter Einsatz der Elementarfaktoren Arbeit, Betriebsmittel und Rohstoffe (Werkstoffe) hergestellt wird. Ergänzend hierzu ist das Erzeugnis gemäß DIN 6789 ein in sich geschlossener und funktionsfähiger Gegenstand, der aus einer Menge von Gruppen und/oder Teilen besteht [DIN03].

Die vorangegangene Definition nach DIN führt zum Begriff der *Komponente* als Bestandteil eines Produkts. Dabei kann eine solche Komponente ihrerseits selbst wieder aus anderen Komponenten zusammengesetzt sein. Dieser rekursive Prozess lässt sich theoretisch beliebig fortsetzen, ist praktisch aber durch die Teilbarkeit einer Komponente begrenzt. Bestandteile eines Produkts, die nicht weiter zerlegbar sind, werden fortan als atomare Komponenten bezeichnet. Angelehnt an [Dre08] werden im Rahmen dieser Arbeit drei Arten von Komponenten unterschieden. Die sogenannten Basiskomponenten stellen die grundsätzlich notwendigen Bestandteile eines Produkts dar, während Implikativkomponenten erst in gewissen Situationen erforderlich werden. Derartige Zustände können beispielsweise durch das Hinzufügen einer nicht zwingend benötigten Optionalkomponente mit eigenen Abhängigkeiten zu anderen Komponenten entstehen.

Die beschriebene Differenzierung von Komponenten, insbesondere die Existenz von Optionalkomponenten, lässt die Definition sogenannter *Varianten* eines Produkts zu, unter denen man nach [BI97] größtenteils übereinstimmende Produkte mit Abweichungen in einzelnen Merkmalen versteht. Gemäß [Bar94] klassifiziert man die daraus resultierende Variantenvielfalt eines Produkts einerseits in die innere bei der Produktion auftretende und andererseits in die äußere für den Kunden erkennbare Vielfalt. Ein Konzept sowohl zur Beherrschung als auch Optimierung dieser Variantenvielfalt während der Produktentwicklung und Produktfertigung (Abschnitt 2.1.2) sowie der Zusammenstellung von Komponenten (Abschnitt 2.2) wird als Variantenmanagement [Fra02] bezeichnet.

Alternativ lässt sich der Produktbegriff auch als Ergebnis oder Zielstellung der eigentlichen *Produktion* auffassen, wobei dieser Prozess wiederum aus unterschiedlichen Perspektiven betrachtet werden kann. Beispielsweise steht laut [KS02] die Kombination und Transformation von Dienstleistungen und Gütern zu anderen Dienstleistungen und Gütern im Vordergrund, während im Sinne der Fertigstellung die eigentliche Ver- und Bearbeitung von Rohstoffen in [Jun06] als Charakter der Produktion angesehen wird, um Halb- oder Fertigfabrikate zu erhalten. Damit schließt sich der Kreis zum Produktbegriff als ursprünglichen Ausgangspunkt der Untersuchungen, im weiteren Verlauf der Arbeit soll ein Produkt durch folgende Zusammenfassung in Anlehnung an [Kru10] definiert sein.

▶ **Definition 2.1** *Ein **Produkt** ist ein aus verschiedenen (atomaren) Komponenten zusammengesetztes Ergebnis eines Herstellungsprozesses und kann materieller oder immaterieller Natur sein, wodurch es synonym für Güter, Waren oder auch Dienstleistungen verwendet werden kann. Dabei dient es entweder der konsumentenbezogenen Bedürfnisbefriedigung oder als Investitionsobjekt zur Herstellung weiterer Produkte. Schließlich lässt es sich nach verschiedenen Dimensionen klassifizieren.*

2.1.1.2 Produktarten

Wie in der vorangegangenen Definition 2.1 bereits verdeutlicht, haben Produkte bedingt durch die üblicherweise komponentenbasierte Zusammenstellung und die sich daraus ergebende Kombinationsvielfalt ganz unterschiedliche charakteristische Eigenschaften. Deren Klassifizierung als Produktarten soll in diesem Abschnitt im Mittelpunkt stehen. Abhängig vom Anwendungskontext bietet hierfür die Fachliteratur gemäß [Kru10] eine Vielzahl von Strategien zur Kategorisierung, im Folgenden sollen zwei für diese Arbeit relevante Ansätze vorgestellt werden.

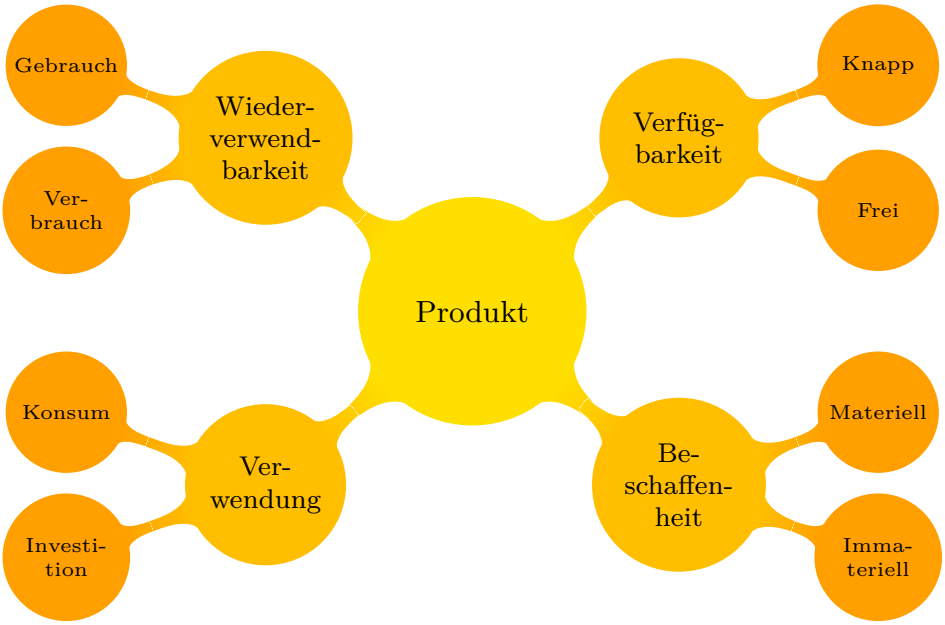


Abbildung 2.1: Klassifizierung von Produktarten nach [DS05]

Ausgehend von den resultierenden Eigenschaften eines Produkts werden diese in [DS05] nach vier Dimensionen klassifiziert, welche gemeinsam mit den jeweiligen Ausprägungen in **Abbildung 2.1** dargestellt sind. Mit der *Verfügbarkeit* als erste Dimension lassen sich freie von knappen Produkten unterscheiden. Dabei kann ein freies Produkt, wie beispielsweise die Atemluft als Ergebnis der pflanzlichen Photosynthese, ohne jegliche Gegenleistung im wirtschaftlichen Sinn einfach konsumiert werden, während knappe Güter, beispielsweise eine Segelyacht, in der Regel eine Vergütung meist in Form von Zahlungsmitteln erfordern. Aufgrund dieser Differenzierung spielen freie Produkte für den Prozess der Zusammenstellung von Komponenten keine direkte Rolle. Eine weitere Dimension gibt die Frage nach der *Beschaffenheit* vor, wobei hier die bereits erwähnten Ausprägungen materieller und immaterieller Produkte existieren. Betrachtet man den Zweck beziehungsweise die *Verwendung* eines Produkts als dritte Dimension, lassen sich Konsumprodukte für die direkte Bedürfnisbefriedigung (wie beispielsweise Nahrungsmittel) von Investitionsprodukten zur Herstellung anderer Produkte unterscheiden. Schließlich legt die letzte Dimension der *Wiederverwendbarkeit* die Art des Konsums fest, wobei Gebrauchsprodukte, beispielsweise ein Fahrrad, für den mehrfachen Einsatz über einen längeren Zeitraum vorgesehen sind. Dagegen lassen sich Verbrauchsprodukte, wie unter anderem die bereits erwähnten Nahrungsmittel, nur einmal konsumieren.

Betrachtet man Produkte dagegen aus externer Kundensicht, lassen sich diese anhand der Kundeneinflussnahme unterscheiden [Sch06]. Dabei kennzeichnen die verfügbaren Freiheitsgrade zur Anpassung und Gestaltung eines Produktes das Ausmaß der in Abschnitt 1.1 kurz erläuterten Individualisierung und führen zu einer klassischen Dreiteilung. Einerseits gibt es Produkte, die bezüglich ihrer Zusammenstellung entweder keinen oder vollständigen Einfluss durch den Kunden erlauben und demzufolge als anbieterorientiert beziehungsweise kundenorientiert bezeichnet werden. Auf der anderen Seite gibt es die sogenannten kundenzentrierten Produkte, die keinem der Extreme zugeordnet werden können und somit einen mehr oder weniger begrenzten Spielraum zur Anpassung bieten.

2.1.2 Produktfertigung

Anknüpfend an die im vorangegangenen Abschnitt vorgestellte Klassifizierung von Produktarten gemäß der Kundeneinflussnahme soll hier ebenfalls das Kriterium des Individualisierungsgrads zur Typisierung der Produktfertigung wie in [Kru10] diskutiert werden. In diesem Zusammenhang kann bezüglich der zu produzierenden Stückzahl prinzipiell die Einzelfertigung der Mehrfachfertigung gegenüber gestellt werden, wobei für letztere wiederum verschiedene Ausprägungen existieren. Ein Vergleich aller nachfolgend erläuterten Typen der Produktfertigung findet sich in **Tabelle 2.1**.

Merkmal / Fertigungstyp	Einzel-fertigung	Serien-fertigung	Varianten-fertigung	Massen-fertigung	indiv. Massen-fertigung
Stückzahl	−−[1]	−[2]	o	++	o
Produktvorgabe	ao	po, ao	po, ao	po	po, ao
Kosten/Stück	++	o	o	−−	−
Flexibilität	++	−	o	−−	+
Automatisierungsgrad	−−	o	o	++	+
Konfigurationseignung	−−	−−	+[3]	−−	++

++: sehr hoch +: hoch o: durchschnittlich −: niedrig −−: sehr niedrig
ao: auftragsorientiert (Kunde) po: programmorientiert (Hersteller)

[1] Einzelstück [2] Klein-/Großserien [3] abhängig von der Definition

Tabelle 2.1: Vergleich von Fertigungstypen nach [Abe04]

2.1.2.1 Einzelfertigung

Unter der Einzelfertigung beziehungsweise Auftragsfertigung versteht man die klassische handwerkliche Herstellung eines Einzelstücks im Sinne eines Unikats, welches sehr individuelle Wünsche und Anforderungen im Rahmen eines konkreten Kundenauftrags erfüllen soll. Mit dieser großen Flexibilität sind jedoch eine schlechte Automatisierbarkeit sowie mitunter auch sehr hohe Kosten in der Fertigung verbunden. Aufgrund jener Charakteristik eignet sich die Einzelfertigung nicht für die Produktkonfiguration. Beispielsweise stellt das Automobil „Maybach Exelero“[1] eine solche Spezialanfertigung dar, welches von der Maybach-Manufaktur in Zusammenarbeit mit Fulda Reifen für die Markteinführung eines neuen Hochgeschwindigkeitsreifens gebaut wurde.

[1] http://de.wikipedia.org/wiki/Maybach_Exelero

2.1.2.2 Serienfertigung

Im Gegensatz zur Fertigung von Einzelstücken beschreibt die Serienfertigung die Herstellung von Produkten mit höheren Stückzahlen. Zusätzlich wird hier zwischen Klein- und Großserien unterschieden, wobei Kleinserien üblicherweise einstellige bis hohe zweistellige Stückzahlen abdecken. Über die einzelnen Serien eines Produkts bleiben dessen Komponentenstruktur und die notwendigen Produktionsanlagen konstant, lediglich die zum Einsatz kommenden Werkstoffe können geringfügige Abweichungen aufweisen. Gegenüber der Einzelfertigung ist dadurch laut [Abe04] eine viel kosten- und ablaufeffizientere Fertigung möglich, die jedoch auch mit einer begrenzteren Flexibilität verbunden ist. Für die Produktkonfiguration ist deswegen die Serienfertigung nicht geeignet, ein Vertreter dieses Typs ist beispielsweise die Herstellung von Autoreifen.

2.1.2.3 Variantenfertigung

Basierend auf der Serienfertigung ist das Ziel der Variantenfertigung eine in Ausprägung und Stückzahl variierende Produktherstellung. Dazu können sich nun über die einzelnen Serien auch die Zusammensetzung der Produktkomponenten oder deren Eigenschaften verändern, womit eine gewisse Variantenvielfalt erreicht wird. Diese soll jedoch ohne nennenswerte Abweichungen bezüglich der Stückkosten und des Automatisierungsgrads gegenüber der Serienfertigung erfolgen. Schließlich lässt sich die Variantenfertigung als geeignet für die Produktkonfiguration einordnen, wenn die kundenspezifische Zusammenstellung der Produktkomponenten vor der eigentlichen Fertigung erfolgt. Andernfalls findet durch den Kunden nur die Auswahl einer unveränderbaren Produktvariante statt. Ein Beispiel, welches in der Praxis in beiden Szenarien anzutreffen ist, stellt die (individuelle) Bedruckung von T-Shirts dar.

2.1.2.4 Massenfertigung

Der vollständig gegensätzliche Prozess zur Einzelfertigung präsentiert sich in der Massenfertigung, die in einer dauerhaften Wiederholung eines fixierten und weitestgehend automatisierten Ablaufs Produkte in einer enorm großen Menge erzeugt. Dadurch sinken die Stückkosten gegenüber allen anderen Fertigungstypen deutlich. Allerdings ist damit auch die geringste Flexibilität zur Produktgestaltung durch den Kunden verbunden, weswegen die Massenfertigung nicht für die Produktkonfiguration geeignet ist. Typische Beispiele stellen Verbrauchsprodukte wie Salz, Mehl oder Nudeln dar.

2.1.2.5 Kundenindividuelle Massenfertigung

Wie einleitend im Abschnitt 1.1 beschrieben, versucht die auch als Mass Customization bezeichnete kundenindividuelle Massenfertigung jene kosten- und ablaufspezifischen Vorteile der Massenproduktion mit der hohen Flexibilität gegenüber Kundenwünschen in der Einzelfertigung zu kombinieren [RP03]. Das Ziel der preiswerten Fertigung einer großen Stückzahl von im zulässigen Rahmen individualisierten Produkten soll unter der Nutzung von Standardisierungs- und Synergieeffekten bezüglich der Komponenten erreicht werden. Dadurch ist gegenüber der Variantenfertigung ein höherer Automatisierungsgrad möglich,

der jedoch auch notwendig ist, um bei ähnlichen Produktionsmengen geringere Stückkosten zu erzielen. Aufgrund der Tatsache, dass die Fertigung eines Produkts erst stattfindet, nachdem dieses vom Kunden spezifiziert oder aus typischen Marktanforderungen abgeleitet wurde, ist die Mass Customization bestens geeignet für die Produktkonfiguration. Hierfür ist die Automobil-Branche als klassischer Anwendungsfall zu nennen.

2.2 Definition

Wie im vorangegangenen Abschnitt 2.1.2 erörtert wurde, sind Produktkonfiguratoren im Rahmen der Variantenfertigung sowie der kundenindividuellen Massenfertigung sinnvoll einsetzbar. Nachfolgend soll auf die Gründe für diese Einschätzung näher eingegangen werden. Dazu gibt Abschnitt 2.2.1 einleitend einen Überblick zu den verschiedenen Definitionsansätzen zur Charakterisierung dieses Werkzeugs. Im weiteren Verlauf werden sowohl notwendige Voraussetzungen für den Einsatz von Produktkonfiguratoren in Abschnitt 2.2.2 als auch damit verbundene Anforderungen und Nutzenpotentiale aus verschiedenen Perspektiven in Abschnitt 2.2.3 betrachtet.

2.2.1 Begriffsprägung

Aufgrund des jeweils spezifischen Anwendungshintergrunds sowie des primären Einsatzzwecks haben sich in der Fachliteratur zum Thema Produktkonfigurator recht unterschiedliche Definitionen etabliert, typische Formulierungen sind basierend auf [Kru10] als Zusammenstellung in **Anhang A** zu finden. Der Begriff selbst stellt aus etymologischer Sicht eine Komposition der Worte *Produkt* (siehe Abschnitt 2.1.1) und *Konfigurator* dar, wobei sich letzteres vom Verb *konfigurieren* (lat. configuratio) ableitet und im Sinne von *einrichten*, *anordnen*, *gestalten* oder *zusammenstellen* verwendet wird.

Grundsätzlich stellen Produktkonfiguratoren ein softwaretechnisches Instrument dar, um den einleitend in Abschnitt 1.1 erläuterten Trend der Individualisierung im Bereich der kundenseitigen Produktnachfrage zu unterstützen bzw. überhaupt erst beherrschbar zu realisieren. Insbesondere soll der Kunde befähigt werden, selbstständig ein Produkt nach seinen Wünschen spezifizieren zu können [Pol08]. Daher vermitteln Produktkonfiguratoren auch im Sinne eines Kommunikationskanals zwischen den Bedürfnissen von Kunden sowie dem Angebot eines Unternehmens und stellen somit ein Bindeglied dar [Göb09]. Die hierbei relevanten Produkte werden aus zuvor definierten Produktkomponenten (siehe Definition 2.1) bezüglich der nutzergewählten Merkmalsausprägungen und Komponenteneigenschaften zusammengesetzt, wobei automatisch die Konsistenz gemäß eines formalisierten Regel- und Abhängigkeitswerks sichergestellt werden kann [Sch06].

Eine weiterführende Charakterisierung von Produktkonfiguratoren wird durch die Klassifikation in Abschnitt 2.3 präsentiert, hier soll zur Abrundung der Begriffsbildung noch auf eine Abgrenzung und Einordnung gegenüber den Produktkatalogen und Produktdesignwerkzeugen eingegangen werden. Betrachtet man den Individualisierungsgrad für die angebotenen Produkte, so lässt sich feststellen, dass Produktkataloge vorrangig die Filterung einer Produktmenge anhand der nutzerspezifischen Selektion gewisser Eigenschaften (beispielsweise der Farbe) unterstützen. Die im Katalog enthaltenen Produkte sind jedoch fest definiert, entsprechend ihrer Merkmalsausprägungen in einem Klassifikationssystem

strukturiert, unveränderbar und typischerweise zum Zeitpunkt der Bestellung bereits hergestellt [UES03]. Dagegen unterstützen Produktdesignwerkzeuge einen viel größeren Grad der Individualisierung durch kundenseitige Beeinflussung der einzelnen Produktkomponenten bezüglich Anzahl, Kombinationsmöglichkeiten sowie inhaltlicher Spezifikation [Sch06]. Dieser direkte Einfluss auf die Definition eines Produkts (siehe Abschnitt 2.1.1) ist im Kontext der vorliegenden Arbeit dem Produktentstehungsprozess zugeordnet und zählt damit nicht zum Aufgabengebiet eines Produktkonfigurators.

2.2.2 Voraussetzungen

Die zur Begriffsprägung aufgeführten Funktionen und Zielstellungen von Produktkonfiguratoren (siehe Abschnitt 2.2.1) bringen für deren Verwirklichung gewisse Voraussetzungen mit sich. Diese umfassen die Modularisierung der zu konfigurierenden Produkte, die Formalisierung von Abhängigkeiten zwischen Produktkomponenten sowie die Transformation von Kundenanforderungen und werden nachfolgend genauer erläutert.

2.2.2.1 Modularisierung von Produkten

Um die wunschgemäße Produktzusammenstellung über die Einzelfertigung hinaus auch im Rahmen der kundenindividuellen Massenfertigung überhaupt anbieten zu können, muss die Struktur der einzelnen Produkte und ihrer Komponenten im Sinne von Definition 2.1 klar bestimmt sein. Diese sogenannte *Modularisierung* erfordert jedoch intellektuelle Vorarbeit zur Festlegung von Strukturierungsebenen, Granularitäten und Begriffen für die konkrete Situation eines Unternehmens. Dabei kann die Zerlegung der Produkte auch mehrstufig (rekursiv) nach [DIN03] erfolgen. Letztlich liegt gemäß [Kru10] die Eignung zur Produktkonfiguration vor, wenn mindestens eine Eigenschaft einer Basiskomponente mehrere Möglichkeiten zur Wertbelegung aufweist oder das Produkt durch wenigstens eine Basiskomponente und eine Optionalkomponente aufgebaut ist. Die Gesamtheit aller derartig identifizierten Komponenten wird nach [Els03] als *Modulbaukasten* bezeichnet.

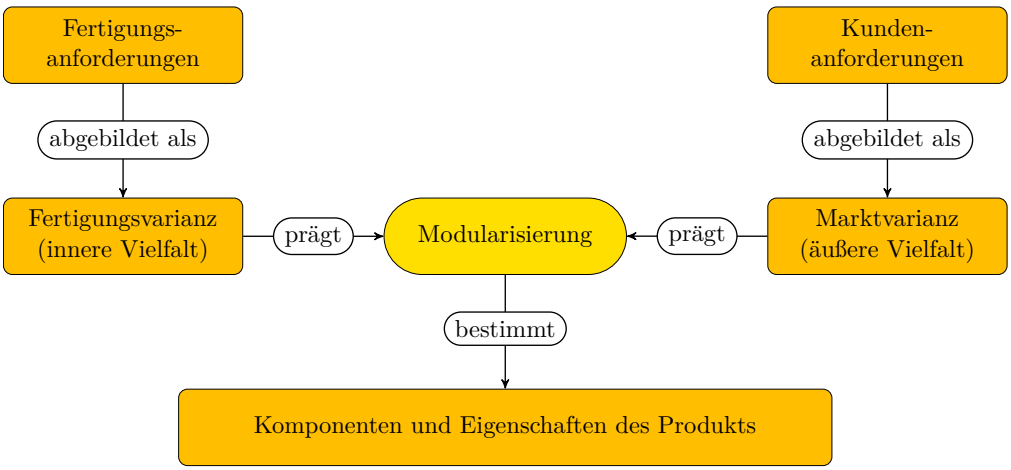


Abbildung 2.2: Einfluss der inneren und äußeren Varianz auf die Modularisierung

Damit das Ergebnis der in **Abbildung 2.2** dargestellten Modularisierung für den nachfolgenden Einsatz im Produktkonfigurator den größtmöglichen Nutzen erzielt, muss dieser

Prozess bereichsübergreifend stattfinden. Dabei beeinflussen insbesondere die von Entwicklung und Produktion vorgegebene innere Varianz sowie die von Marketing und Vertrieb gegenüber den Kundenanforderungen (siehe Abschnitt 2.2.3) verantwortete äußere Varianz eines Produkts dessen Modularisierung. Aus unternehmerischer Sicht besteht jedoch das Ziel, die innere Fertigungsvarianz aus Kostengründen möglichst zu reduzieren und gleichzeitig die äußere Marktvarianz sowie die damit verbundene Abdeckung kundenseitiger Wünsche zu maximieren [Wüp00]. Als Lösungsbeitrag für dieses Optimierungsproblem hat sich die *Gleichteilestrategie* zur Verringerung der inneren Varianz unter Nutzung von standardisierten Baugruppen in verschiedenen Produkten und Größenstufungen durchgesetzt [Pli12]. Beispielsweise wird seit vielen Jahren das Plattform-Konzept im Automobilbau verwendet, um markenübergreifend bei der Herstellung verschiedener Modelle Synergieeffekte und Kosteneinsparungen innerhalb eines Konzerns zu erzielen.

2.2.2.2 Formalisierung von Abhängigkeiten

Parallel zur Modularisierung sind für die resultierenden Komponenten eines Produkts weitere Informationen zu existierenden Abhängigkeiten zu erfassen. Die Menge aller Abhängigkeitsbeschreibungen aus technischen, kostenbedingten oder auch vertriebsrelevanten Gründen wird als *Wissensbasis* bezeichnet und definiert für die Komponenten des Modulbaukastens, in welchen Beziehungen diese untereinander stehen. Erst durch eine solche Formalisierung sind Produktkonfiguratoren in der Lage, die individuelle Zusammenstellung eines Produktes gemäß der vom Anbieter gewünschten Semantik zu überwachen und zu unterstützen. Daneben hat die Erstellung einer Wissensbasis auch das Ziel, implizit vorhandenes und teilweise stark verteiltes Wissen der Mitarbeiter über Produktstrukturen zu vereinheitlichen und für andere Personen, Bereiche oder Anwendungen zugänglich zu gestalten. Typische Abhängigkeiten sind nach [Kru10] beispielsweise

- welche Basiskomponenten miteinander kombinierbar sind und welche Komponenten sich gegenseitig ausschließen,
- welche Komponenten bezüglich der Basiskomponenten als Optionalkomponenten charakterisierbar sind,
- welche Komponenten bezüglich der Basis- und Optionalkomponenten als Implikativkomponenten charakterisierbar sind oder
- welche Wertbelegungsmöglichkeiten für welche Komponenten in den verschiedenen Kombinationen zulässig sind.

Für die Spezifikation derartiger Abhängigkeiten stehen unterschiedliche Techniken und Konstrukte zur Verfügung, unter anderem Wenn-Dann-Regeln und Entscheidungstabellen [Irr95]. Dadurch definiert der Produktanbieter einen Lösungsraum, der alle potentiell realisierbaren Kombinationen von Komponenten, welche auch als *Produktkonfigurationen* bezeichnet werden, umfasst [Pol08]. Obwohl für eine korrekte Interpretation durch den Produktkonfigurator die formale Widerspruchsfreiheit der Wissensbasis gewährleistet sein muss, lässt sich diese Eigenschaft in der Praxis aufgrund der Komplexität möglicher Kombinationen häufig nicht bereits zum Zeitpunkt der Definition statisch überwachen. Stattdessen kann diese Anforderung erst während der Prozessierung der Abhängigkeiten durch zusätzliche Mechanismen wie Zyklenerkennung dynamisch geprüft werden. Auf weitere Details hierzu wird im Rahmen der Klassifikation in Abschnitt 2.3 eingegangen.

2.2.2.3 Transformation von Kundenwünschen

Kundenseitige Anforderungen beeinflussen wie bereits erläutert den Prozess der Modularisierung eines Produkts über dessen äußere Varianz. Allerdings werden die im Konfigurator angebotenen Kombinationsmöglichkeiten für ein konkretes Produkt nie die vollständige Menge der Kundenwünsche abdecken, vor allem weil einerseits eine hohe Marktvarianz der angestrebten geringen inneren Varianz entgegenwirkt und andererseits die individuellen Anforderungen jedes potentiellen Kunden in aller Regel nicht effizient ermittelbar sind. Deswegen stellt die im Zuge der Modularisierung festzulegende Benutzerführung durch den Konfigurationsprozess die wichtigste Schnittstelle zum Kunden dar, um dessen konkrete Anforderungen optimal auf eine passende Produktvariante abzubilden.

Eine derartige Benutzerführung erfolgt in den meisten Produktkonfiguratoren nach dem Frage-Antwort-Prinzip, d.h. durch Beantwortung von mehr oder weniger technischen Fragen mit einer definierten Menge von Antwortmöglichkeiten werden die Bedürfnisse des Kunden ermittelt, im Hintergrund die relevanten Produktkomponenten ausgewählt sowie gegebenenfalls die nachfolgenden Fragen bestimmt. Dabei zeigen Studien, dass eine wachsende Zahl an Alternativen für eine Entscheidung nicht nur zu mehr Verwirrung und Präferenzlosigkeit führen [PKMS03, Sch06], sondern auch die Gefahr der Demotivation und Unzufriedenheit mit der getroffenen Auswahl steigt [IL00]. Deswegen sollte mit möglichst wenigen Fragen der Konfigurationsprozess durchführbar sein, wobei jedoch das individuelle Hintergrundwissen der Nutzer beachtet werden muss. Zudem kennen Kunden oft ihre tatsächlichen Anforderungen nicht und möchten diesbezüglich oder bei der Formulierung existierender Wünsche durch den Konfigurator unterstützt werden [BAKF04].

2.2.3 Anforderungen

Neben allgemeinen Erwartungen an Softwaresysteme lassen sich speziell für Produktkonfiguratoren Anforderungen seitens der Kunden und Anbieter wie in **Abbildung 2.3** dargestellt unterscheiden. Diese werden nachfolgend basierend auf [BAKF04, Dre08, Göb09, Kru10, Sch06, Wüp00] näher erläutert.

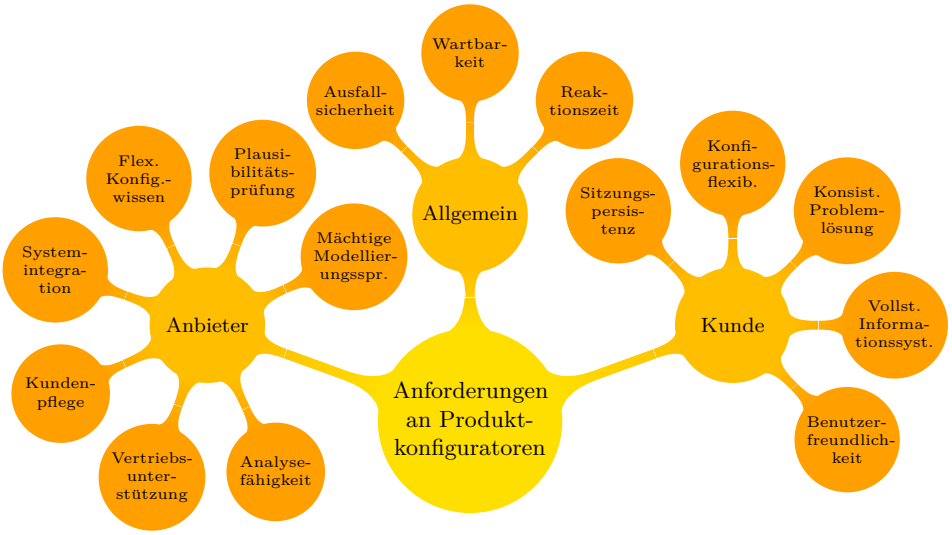


Abbildung 2.3: Anforderungen an Produktkonfiguratoren

2.2.3.1 Allgemein

Reaktionszeit: Aufgrund der enormen Leistungssteigerungen über die vergangenen Jahrzehnte im Bereich der Prozessoren und des Speichers können Computer immer komplexere Aufgaben in immer kürzerer Zeit bewältigen. Dadurch wurde die ursprüngliche Stapel- bzw. Batchverarbeitung durch eine zunehmend interaktivere Arbeitsweise abgelöst. Insbesondere im Umfeld der webbasierten Anwendungen, zu denen vor allem E-Commerce-Systeme und Produktkonfiguratoren zählen, sind bereits Veränderungen der Reaktionszeit um Zehntelsekunden entscheidend für die Akzeptanz. Beispielsweise liegt die Erwartung eines durchschnittlichen Online-Shoppers bei maximal zwei Sekunden für den vollständigen Aufbau einer Webseite [Aka09]. Je größer die Latenz und damit Wartezeit für den Kunden wird, desto höher ist somit das Abbruchrisiko des aktuellen Konfigurationsvorgangs [Sch06].

Wartbarkeit: Die Wartung eines Produktkonfigurators erstreckt sich über zwei Ebenen. Auf der einen Seite werden aus softwaretechnischer Sicht in gewissen Abständen neue Versionen und Aktualisierungen der Konfigurationssoftware bereitgestellt, um beispielsweise Fehler zu beseitigen oder neue Funktionen zu integrieren. Andererseits unterliegt auch die Wissensbasis sich verändernden Anforderungen und muss diesen entsprechend angepasst werden. Im Idealfall sollten Anbieter in beiden Fällen die notwendigen Maßnahmen ohne großen Aufwand durchführen können, sofern eine geeignete Infrastruktur und das notwendige Expertenwissen aufgebaut sind. Alternativ lassen sich diese Aufgaben auch an einen Dienstleister auslagern.

Ausfallsicherheit: Neben den immer kürzeren Reaktionszeiten ist durch die mittlerweile starke Vernetzung unserer Welt auch der Anspruch auf eine ständige Verfügbarkeit webbasierter Anwendungen gewachsen. Dieser kann nur durch eine sehr hohe Ausfallsicherheit des Gesamtsystems erfüllt werden, welche häufig mittels einer redundanten Auslegung von Teilkomponenten erreicht wird. Beispielsweise kann der Aufruf eines Produktkonfigurators im Inter- bzw. Intranet mit Hilfe eines Lastverteilungssystems transparent für den Nutzer auf einen von parallel existierenden Application-Servern geleitet werden. Bei Ausfall eines einzelnen Servers lassen sich die darauf arbeitenden Nutzer auf ein benachbartes System umlenken. Die Unterstützung jenes sogenannten *Session-Failover* und eines generell verteilten Anwendungsszenarios sind bei der Entwicklung des Produktkonfigurators entsprechend zu berücksichtigen.

2.2.3.2 Kunde

Benutzerfreundlichkeit: Gemäß [RP03] sind für den Erfolg des webbasierten Vertriebs von Produkten nicht primär deren marketingrelevanten Eigenschaften von Bedeutung, sondern stattdessen das zur Produktpräsentation gegenüber dem Kunden eingesetzte Werkzeug. Demzufolge spielt die Benutzerfreundlichkeit eine entscheidende Rolle für die Akzeptanz eines Produktkonfigurators. Unter diesem Kriterium wird neben einer intuitiven Bedienbarkeit über eine klare und barrierefrei strukturierte Oberfläche mit einer durchgängig konsistenten Navigation auch die gefühlte Erlebbarkeit der Produktkonfiguration als ein angenehmer und zufriedenstellender Prozess für den Anwender verstanden [Pol08]. Hierzu ist sowohl die internationalisierte Präsentation wie in [Göb09] diskutiert als auch die breite Unterstützung der verschiedenen kundenseitig genutzten Browser und Endgeräte notwendig. Das Sicherstellen

einer hohen Benutzerfreundlichkeit ist umso wichtiger, da im Problemfall einem Kunden die Unterscheidung zwischen inhaltlichen Fehlern in der angebotenen Produktpalette und Fehlern in der umgebenden Konfigurator-Software typischerweise nicht möglich ist [Dre08].

Vollständiges Informationssystem: Die individuelle Zusammenstellung eines Produkts erfolgt auf Grundlage von Entscheidungen während des Konfigurationsprozesses. Die präferenzgesteuerte Auswahl zwischen verschiedenen Alternativen ist für Kunden jedoch nur dann möglich, wenn für einen Vergleich ausreichend Informationen bereitgestellt werden. Um einen Produktkonfigurator als zentralen Kommunikationskanal zum Kunden zu etablieren, muss die Menge an Informationen, wie beispielsweise die technischen Eigenschaften und Preise von Komponenten oder der Gesamtpreis des aktuell zusammengestellten Produkts, sinnvoll integriert und nachvollziehbar dargestellt werden.

Konsistente Problemlösung: Neben der Bereitstellung aller notwendigen Informationen erwarten Kunden bei der Durchführung eines Konfigurationsvorgangs auch eine Hilfestellung durch den Produktkonfigurator. Dieser soll basierend auf den vom Nutzer formulierten Anforderungen mögliche Vorschläge und deren Auswirkungen auf die Lösungen des Zusammenstellungsproblems unterbreiten, wobei stets eine inhaltliche und technische Zuverlässigkeit nach [Dre08] gegeben sein muss. Ein typisches Beispiel hierfür sind Konfliktdialoge. Die in diesem Prozess notwendigen Entscheidungen müssen auch ohne das zugrunde liegende Expertenwissen zu den Komponenten und deren Abhängigkeiten vom Anwender getroffen werden können.

Konfigurationsflexibilität: Die Durchführung eines Konfigurationsvorgangs entspricht in den wenigsten Fällen einem zuvor vollständig durchgeplanten Vorgang. Stattdessen dient ein Produktkonfigurator auch immer der Erkundung und dem Ausprobieren möglicher Alternativen in der angebotenen Produktpalette (siehe Benutzerfreundlichkeit). Demzufolge kann durch den Kunden der Wunsch zum Rücksetzen bzw. Umbewerten einer bereits getroffenen Entscheidung bezüglich der Anwahl oder Abwahl einer Komponente zu einem späteren Zeitpunkt entstehen. Diese Flexibilität im Konfigurationsprozess setzt voraus, dass direkte oder indirekte Auswirkungen der Änderung soweit möglich in die aktuelle Zusammenstellung integriert werden, wobei davon unabhängige eigenschafts- oder komponentenspezifische Entscheidungen erhalten bleiben sollten. Weiterhin besteht die Aufgabe, die hohe Komplexität dieser Anpassungsfähigkeit durch den Produktkonfigurator so abzubilden, dass dem Nutzer dennoch eine weitestgehend intuitive Konfiguration ermöglicht wird.

Sitzungspersistenz: Ein komplexes und umfangreiches Produktangebot kombiniert mit der explorativen Nutzungsmöglichkeit von Produktkonfiguratoren führen neben anderen Faktoren dazu, dass eine Produktzusammenstellung üblicherweise nicht im ersten Schritt abgeschlossen und zur Bestellung geführt wird. Vielmehr erwartet ein Kunde, dass sein aktueller Konfigurationszustand speicherbar, zu einem späteren Zeitpunkt wieder aufrufbar und die Konfiguration basierend auf dem zuvor gespeicherten Zustand fortsetzbar ist, ohne dass dabei Informationen über bereits vorgenommene Entscheidungen verloren gehen.

2.2.3.3 Anbieter

Mächtige Modellierungssprache: Aufgrund des Individualisierungsgrads sowie der Modularisierungsstrategie für eine Produktgruppe können neben einer Vielzahl an Komponenten auch komplexe Abhängigkeitsbeschreibungen notwendig werden. Deswegen muss das Pflegetool des Produktkonfigurators eine entsprechend mächtige Modellierungssprache zur Verfügung stellen, um dem Produktanbieter die Entwicklung des passenden Modulbaukastens zu ermöglichen. Gleichzeitig soll diese Beschreibungssprache aber auch eine intuitive Bedienbarkeit und leichte Erlernbarkeit mit sich bringen, um eine große Akzeptanz bei den verantwortlichen Nutzern mit einem typischerweise eher fachlichen als programmiertechnischen Hintergrund zu erreichen. Die Gestaltung einer mächtigen sowie intuitiv zu bedienenden Modellierungssprache stellt bei der Entwicklung eines Produktkonfigurators eine der zentralen Herausforderungen dar.

Plausibilitätsprüfung: Der Einsatz eines Produktkonfigurators dient unter anderem der Beherrschung von Komplexität im Rahmen der individuellen Produktzusammenstellung. Dabei sollen definierte Regeln die Korrektheit und Baubarkeit einer Konfiguration sicherstellen, d.h. Nutzer können erwarten, dass ihre Selektionen immer zu herstellbaren Produkten führen. In einem komplexen Modulbaukasten ist die Prüfung auf inhaltliche Konsistenz aller Regeln und ihrer Auswirkungen im Zusammenspiel mit anderen Regeln manuell nicht mehr effektiv durchführbar. Zumindest die formale Widerspruchsfreiheit, beispielsweise hinsichtlich Zyklenfreiheit, sollte durch automatische Plausibilitätsprüfungen idealerweise bereits zum Zeitpunkt der Modellierung oder spätestens während des Zusammenstellungsprozesses durch den Produktkonfigurator unterstützt werden.

Flexibles Konfigurationswissen: Durch die Weiterentwicklung von Produkten muss deren initiale Abbildung im Modulbaukasten über den Produktlebenszyklus hinweg angepasst werden können. Dies betrifft sowohl die Charakteristik von Komponenten als auch deren Abhängigkeitsbeschreibungen. Analog zum Kriterium der Wartbarkeit sollte ein Produktkonfigurator die Ausführung der notwendigen Änderungen durch den Anbieter selbst oder einen Dienstleister in effizienter Weise unterstützen.

Systemintegration: Für den sinnvollen Einsatz eines Produktkonfigurators ist dessen Einbettung in bereits existierende Prozesse und Anwendungssysteme notwendig. Neben der Kommunikationsfunktion zwischen den Unternehmensbereichen Entwicklung, Vertrieb und Produktion [Lie11c] spielt vor allem die automatisierbare Durchgängigkeit der Datenversorgung eine große Rolle. Hierunter versteht man einerseits den Aufbau des Modulbaukastens unter Nutzung bereits vorhandener Daten, typischerweise aus Systemen für das Produktdatenmanagement (PDM), um dadurch die aufwändige und fehleranfällige manuelle Erfassung zu vermeiden. Andererseits können während der Konfiguration zusätzliche Informationen dynamischer Natur zur Anzeige kommen, wie beispielsweise die Lieferzeiten oder Einzelpreise von Komponenten. Schließlich muss auch die Weiterverarbeitung abgeschlossener Konfigurationen, beispielsweise das Auslösen einer Bestellung, in den angrenzenden Systemen wie dem Enterprise Resource Planning (ERP) oder Customer Relationship Management (CRM) gewährleistet sein. Zur Unterstützung dieser Funktionalitäten sollten Produktkonfiguratoren entsprechende Schnittstellen oder Erweiterungen vorsehen.

Kundenpflege: Bedingt durch die seit vielen Jahren wachsende Verbreitung und Akzeptanz des elektronischen Handels über das Internet müssen Unternehmen mehr und mehr auch diesem Medium angepasste Methoden zur Kommunikation mit den Kunden einsetzen. Produktkonfiguratoren sollen diesbezüglich aus Sicht der Anbieter nicht nur die individualisierte Zusammenstellung von Produkten ermöglichen, sondern darüber hinaus auch Funktionen zur Kundenpflege erfüllen. Hierzu werden laut [Dre08] beispielsweise die Akquisition von Kunden durch einen überzeugenden Erlebnisfaktor (siehe Benutzerfreundlichkeit), die Intensivierung existierender Kundenbindungen und eine bestmögliche Beratung durch Bereitstellung zusätzlicher Erläuterungen (siehe Vollständiges Informationssystem) gezählt.

Vertriebsunterstützung: Neben der Nutzung im Marketing bieten sich Produktkonfiguratoren auch für die Unterstützung des Vertriebs im traditionellen Vertreter- oder Filialnetz an. Die Anzeige zusätzlicher Informationen zu Produkten und deren Komponenten sowie die Bereitstellung interner Dokumente, beispielsweise von technischen Datenblättern, sollte dabei sinnvollerweise auf Basis der Rollenzuordnung des angemeldeten Nutzers entschieden werden. Dadurch ist es möglich, einen gemeinsamen Applikationsstand für unterschiedlich autorisierte Nutzergruppen zu betreiben.

Analysefähigkeit: Zur Etablierung des Produktkonfigurators als vollwertigen Kommunikationskanal zwischen Anbieter und Kunden gehört neben der Bereitstellung einer Menge von individualisierbaren Produkten auch die Fähigkeit zur Aufzeichnung des darauf stattfindenden Nutzerverhaltens. Dessen Analyse kann Aufschluss über die Akzeptanz des angebotenen Produktspektrums, die Güte der Bedienbarkeit sowie Empfehlungen zur Verbesserung liefern. Relevante Informationen sind in diesem Zusammenhang beispielsweise, welche Komponenten und Kombinationen am seltensten gewählt wurden („Ladenhüter“) oder in welchem Stadium ein Konfigurationsvorgang häufig abgebrochen wird.

2.3 Klassifikation

Nachdem im vorherigen Abschnitt 2.2 sowohl Ansätze zur Definition als auch Anforderungen an Produktkonfiguratoren beschrieben wurden, sollen nun deren charakteristische Merkmale basierend auf den Vorschlägen in [Kru10, Lie11c] klassifiziert werden. Diese Eigenschaften lassen sich gemäß der thematischen Zuordnung in unterschiedliche Klassifikationskategorien gruppieren, deren Herleitung nachfolgend anhand von prinzipiellen Fragestellungen im Zuge der Einführung eines Produktkonfigurators motiviert wird.

Voraussetzung für den Einsatz von Produktkonfiguratoren ist die Klärung der damit aus Unternehmenssicht verbundenen *Geschäftsbasis* (siehe Abschnitt 2.3.1). Zur Abschätzung potentieller Kostenfaktoren und Nutzeneffekte muss unter anderem analysiert werden, welche Produkte für welche Zielgruppe zur Konfiguration geeignet sind und wie die Integration in relevante Geschäftsprozesse möglich ist. Dabei wird die Einbettung in die existierende Systemlandschaft durch die *Technologie* (siehe Abschnitt 2.3.2) des Konfigurators entscheidend mitbestimmt. Diese Kategorie umfasst informationstechnische Kriterien wie die Implementierung, Architektur oder Fragen zur Persistierung eines Anwendungssystems. Unabhängig von der konkreten herstellerspezifischen Ausprägung eines Produktkonfigurators, muss dieser definierte *Prozesse* (siehe Abschnitt 2.3.3) sowohl zur Pflege als auch zur

Verarbeitung für die im Konfigurator benötigten *Daten* (siehe Abschnitt 2.3.4) anbieten. Entsprechend ihrer Entstehung und Verwendung lassen sich diverse Klassen von Daten unterscheiden, wobei die Modelldaten als Abbildung der konkreten unternehmensspezifischen Produktstrukturen die wichtigste Gruppe darstellen. Deren Gestaltung im Sinne eines Modulbaukastens wird durch die Kategorie der *Modellbildung* (siehe Abschnitt 2.3.5) beeinflusst, deren Merkmale unter anderem die Art und den Zeitpunkt der Prüfung des modellierten Beziehungswissens definieren. Schließlich fasst die *Präsentation* (siehe Abschnitt 2.3.6) all jene Kriterien zusammen, welche das Erscheinungsbild des Konfigurationsprozesses gegenüber dem Nutzer bestimmen, wie beispielsweise Festlegungen zum Startzustand, die Darstellung von nicht mehr wählbaren Optionen oder auch die Generierung von Vorschlägen.

Die genannten Klassifikationskategorien und deren zugehörige Merkmale lassen sich wie in **Abbildung 2.4** dargestellt in Form eines Klassifikationssterns übersichtlich anordnen. Dieser wird im weiteren Verlauf des Abschnitts näher erläutert. Dabei erfolgt die Beschreibung der einzelnen Kategorien und zugehörigen Merkmale mit ihren konkreten Ausprägungen jeweils im Uhrzeigersinn, beginnend bei der Geschäftsbasis.

Abbildung 2.4: Kategorien und Merkmale zur Klassifikation von Produktkonfiguratoren

2.3.1 Geschäftsbasis

Die Kategorie der Geschäftsbasis umfasst jene Merkmale, welche im Zusammenhang mit dem Thema Produktkonfiguration direkten Einfluss auf die Unternehmensziele haben bzw. durch diese geprägt werden. Dabei spielen sowohl das Einsatzgebiet und die Zielgruppe als auch die Produktart eine wesentliche Rolle für die Arbeitsweise und das Erscheinungsbild des Konfigurators. Dessen Potential zur Beeinflussung von existierenden Geschäftsprozessen definiert sich zudem über seinen Integrationsgrad. In den folgenden Abschnitten werden die genannten Merkmale näher erläutert.

2.3.1.1 Einsatzgebiet

Sowohl die inhaltliche Aufbereitung der zu modellierenden Produktdaten (siehe Abschnitt 2.3.5) als auch die visuelle Gestaltung (siehe Abschnitt 2.3.6) eines Produktkonfigurators werden maßgeblich durch dessen vorgesehenes Einsatzgebiet geprägt, welches sich primär durch die miteinander agierenden Marktteilnehmer definiert. Diese können typischerweise in private Haushalte, Unternehmen und Verwaltungseinrichtungen differenziert werden, deren mögliche Beziehungen untereinander in **Abbildung 2.5** dargestellt sind. Ausgehend von konkurrierenden Unternehmen, die Produktkonfiguratoren wie in Abschnitt 2.2.3 aufgezeigt als Marketing- und Vertriebskanal mit dem Ziel der Umsatzsteigerung einsetzen, beschränkt sich allerdings deren praxisrelevanter Einsatz auf die Beziehungen zu Privat- und Geschäftskunden eines Unternehmens.

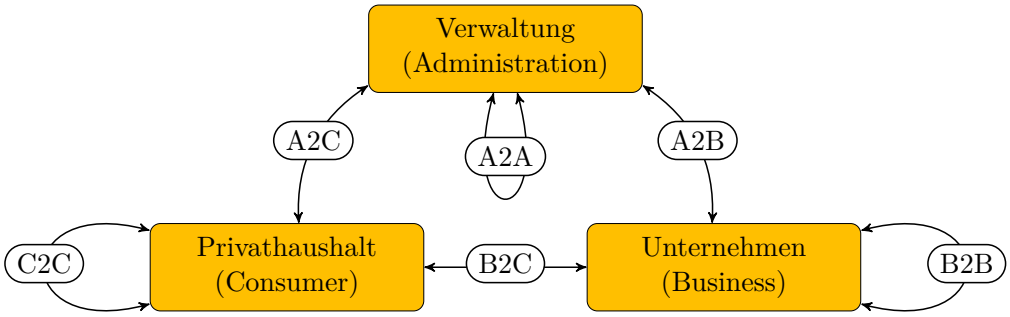


Abbildung 2.5: Marktteilnehmer und deren Beziehungen nach [Kru10]

Das Segment der *Privatkunden*, im Englischen als Business-to-Consumer (B2C) bezeichnet, ist gekennzeichnet einerseits durch Unternehmen als Hersteller bzw. Anbieter von Produkten und andererseits durch Endkunden als Verbraucher bzw. Konsumenten derselben. Ein typisches Beispiel für diese Beziehungsart stellt der Verkauf von Produkten des täglichen Bedarfs wie Grundnahrungsmittel oder Kleidung dar. Dagegen umfasst der im Englischen als Business-to-Business (B2B) benannte Bereich der *Geschäftskunden* alle Beziehungen zwischen Unternehmen, die entweder als Hersteller oder Abnehmer von Produkten agieren. Beispielsweise stehen die Automobil-Produzenten mit ihren verschiedenen Zuliefer-Firmen in derartigen Konstellationen. Die weiteren in Abbildung 2.5 skizzierten Beziehungsmöglichkeiten wie Consumer-to-Consumer (C2C) oder Administration-to-Consumer (A2C) werden durch die Fokussierung von Produktkonfiguratoren auf unternehmensrelevante Prozesse hier nicht näher betrachtet.

2.3.1.2 Zielgruppe

Neben der Verwendung als Kommunikationsmittel zwischen einem Unternehmen und dessen Privat- oder Geschäftskunden lassen sich Produktkonfiguratoren auch durch Mitarbeiter für interne Unternehmensprozesse nutzen, beispielsweise zur Validierung einer kundenspezifischen Konfiguration in der Auftragsbearbeitung. Diese Klassifizierung der Zielgruppe in Kunden und Mitarbeiter hat, wie auch das Merkmal Einsatzgebiet, direkte Auswirkung auf die inhaltliche und visuelle Ausprägung einer Anwendung, welche nach [Sch06] als *Back-End-Systeme* und *Front-End-Systeme* bezeichnet werden. Dabei ist beispielsweise der Umfang erläuternder Produktbeschreibungen abhängig von der Zielgruppe. Typischerweise verfügen Mitarbeiter über das notwendige Expertenwissen zur Produktstruktur, während für Kunden am Front-End-System sowohl die Produkte selbst als auch Entscheidungen des Konfigurators im Rahmen der Zusammenstellung verständlich aufbereitet werden sollten.

2.3.1.3 Produktart

Untrennbar mit der Definition des Einsatzgebiets und der Zielgruppe ist auch die Festlegung der zu konfigurierenden Produkte verbunden. Diese lassen sich prinzipiell wie in Abschnitt 2.1.1.2 erläutert bezüglich der vier Kriterien Verfügbarkeit, Beschaffenheit, Verwendung sowie Wiederverwendbarkeit klassifizieren, wobei jedoch häufig allein aufgrund der Beschaffenheit zwischen *materiellen* und *immateriellen* Produkten unterschieden wird. Allerdings prägt die gesamte Charakteristik eines Produkts tatsächlich dessen in Abschnitt 2.3.5 beschriebene Modellierung.

2.3.1.4 Integrationsgrad

Die Einbettung eines Produktkonfigurators in eine bestehende Systemlandschaft eines Unternehmens betrifft sowohl den Datenaustausch als auch die Nutzung von Funktionalität. Wie in **Abbildung 2.6** dargestellt, lassen sich zudem für beide Dimensionen jeweils zwei Interaktionsrichtungen definieren. Konkrete Beispiele aus Sicht eines Produktkonfigurators sind unter anderem das Importieren von Produktdaten aus einem ERP-System, das Exportieren von Stücklisteninformationen des zusammengestellten Produkts, die Nutzung einer Funktion im CRM-System für kundenspezifische Preise oder die Bereitstellung einer extern nutzbaren Konsistenzprüfung für Komponenten-Zusammenstellungen.

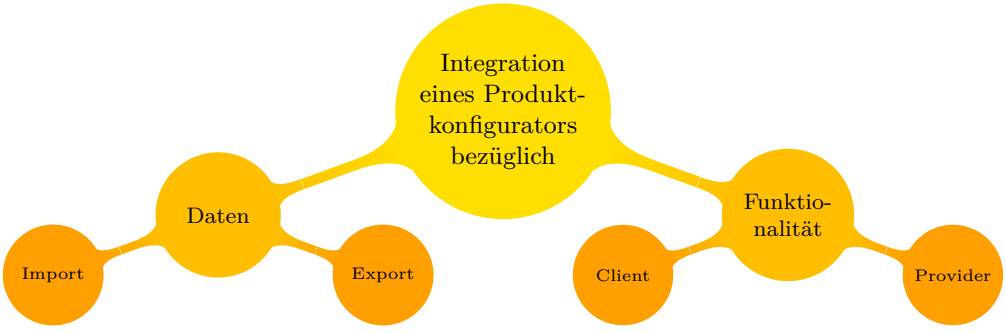


Abbildung 2.6: Dimensionen der Integration für Produktkonfiguratoren

Der Integrationsgrad eines Produktkonfigurators lässt sich gemäß Abdeckung der genannten Dimensionen einem der folgenden drei Varianten zuordnen. Im *nicht integrativen* Fall werden sowohl Daten als auch Funktionen weder aus den existierenden Systemen genutzt noch diesen zur Verfügung gestellt, der Produktkonfigurator wird dann auch als „Standalone-System“ bezeichnet und betrieben. Dafür müssen alle zur Konfiguration erforderlichen Daten vollständig gepflegt und jegliche Funktionen direkt implementiert werden, wodurch das Potential für Inkonsistenzen und Redundanz wächst. Findet dagegen die Nutzung und/oder Bereitstellung von Daten aus bzw. zu den existierenden Informations- und Verwaltungssystemen statt, liegt ein *datenintegrativer* Fall vor. Häufig ist dieser Austausch mit einer Transformation der Daten an den jeweiligen Schnittstellen verbunden. Typisch für dieses Szenario ist die Möglichkeit zur Vermeidung einer redundanten Datenpflege, welche jedoch mit einer verstärkten Abhängigkeit zwischen den Systemen einhergeht. Schließlich erfolgt im *anwendungsintegrativen* Fall der Aufruf von externen bzw. die Kapselung von internen Funktionen inklusive Übergabe der zugehörigen Daten. Hierdurch kann nun auch Redundanz auf funktionaler Ebene verringert werden, wobei sich die gegenseitige Abhängigkeit zwischen Produktkonfigurator und umgebenden Systemen mit entsprechenden Konsequenzen in Ausfallsituationen weiter erhöht.

2.3.2 Technologie

Die Technologie-Kategorie dient der Klassifizierung von Produktkonfiguratoren bezüglich technischer Details. Dabei nehmen Merkmale wie die Universalität oder Implementierung Einfluss auf einzelne Kriterien der Geschäftsbasis (siehe Abschnitt 2.3.1) oder werden von diesen in ihren Ausprägungen eingeschränkt. Weitere ausschlaggebende Faktoren im Rahmen der Einführung eines Produktkonfigurators stellen dessen Architektur und Persistierung dar. Nachfolgend werden die einzelnen Merkmale genauer vorgestellt.

2.3.2.1 Universalität

Hinsichtlich ihrer Einsetzbarkeit in verschiedenen Anwendungsdomänen lassen sich Produktkonfiguratoren nach [Göb09] entweder *Spezialsystemen* oder *Universalsystemen* zuordnen. Erstgenannte sind gekennzeichnet durch die Ausrichtung auf ein spezifisches Produkt oder eine abgegrenzte Produktmenge eines Herstellers, was sich in der alternativen Bezeichnung als Einproduktkonfiguratoren [Sch06] widerspiegelt. Die Zusammenstellung anderer Produkte des gleichen oder eines anderen Herstellers ist nicht vorgesehen bzw. nicht möglich. Im Gegensatz dazu ist die zweite Gruppe der Universalsysteme zur Konfiguration beliebig unterschiedlicher Produkte ohne Fixierung auf ein bestimmtes Unternehmen oder Produktsegment geeignet, weswegen sie auch als Mehrproduktkonfiguratoren bezeichnet werden können. Vertreter jener Kategorie sind beispielsweise der Variantenkonfigurator von SAP oder auch das Produkt CREALIS der ORISA Software GmbH.

2.3.2.2 Implementierung

Für die Implementierung bzw. Realisierung eines Produktkonfigurators gibt es aus Sicht des betreffenden Unternehmens typischerweise drei verschiedene, auch bei anderen Anwendungsklassen nutzbare Vorgehensweisen, welche angelehnt an [Sch06] wie folgt charakterisiert werden können. Einerseits gewährleistet eine kundenspezifische *Projektsoftware* die

größtmögliche Flexibilität bezüglich der existierenden Anforderungen und Wünsche, meist ist diese Form der Software-Entwicklung aber auch mit höheren Aufwänden und Kosten verbunden. Zudem sind im Fall einer unternehmensinternen Eigenentwicklung entsprechend kompetente Mitarbeiter notwendig. Dem gegenüber steht die Nutzung einer kostengünstigeren *Standardsoftware*, welche jedoch nur in begrenztem Maße für spezifische Wünsche erweiterbar ist und deren Modellierungsstrukturen mit den abzubildenden Produkten in Einklang gebracht werden müssen. Darüber hinaus ist die Schulung von ausgewählten Mitarbeitern im Umgang mit dieser Software häufig unumgänglich. Die dritte Möglichkeit der *Weiterentwicklung eines Referenzsystems* erweist sich als Kompromiss aus den beiden bereits genannten Alternativen. Ausgehend von einem bereits existierenden Referenzsystem lässt sich dieses hinsichtlich der unternehmensspezifischen Anforderungen spezialisieren und erweitern. Neben der Voraussetzung, dass der Quellcode jenes Referenzsystems frei verfügbar sein muss, erfordert auch die Einarbeitung der Anwender nicht unerheblichen Aufwand. Jedoch lässt sich in den letzten Jahren eine zunehmende Verbreitung dieses „Open Source“-Gedankens[2] beobachten.

2.3.2.3 Architektur

Konfigurationssysteme bestehen im Allgemeinen aus einem Pflegesystem zur Definition des Modulbaukastens und dem eigentlichen Konfigurator zur Durchführung der Produktzusammenstellung. Für deren Zusammenspiel sowie Verteilung zwischen den Ressourcen von Anbieter und Kunde lassen sich zwei elementare Möglichkeiten unterscheiden: eine *zentrale* und eine *dezentrale* Architektur. Mit Fokus auf die für Anwender relevante Konfiguration können gemäß [Lie11c] zwei grundsätzliche Szenarien, wie in **Abbildung 2.7** dargestellt, definiert werden. Nachfolgend findet deren Beschreibung statt, auf die prinzipiell in Produktkonfiguratoren ablaufenden Prozesse wird in Abschnitt 2.3.3 eingegangen.

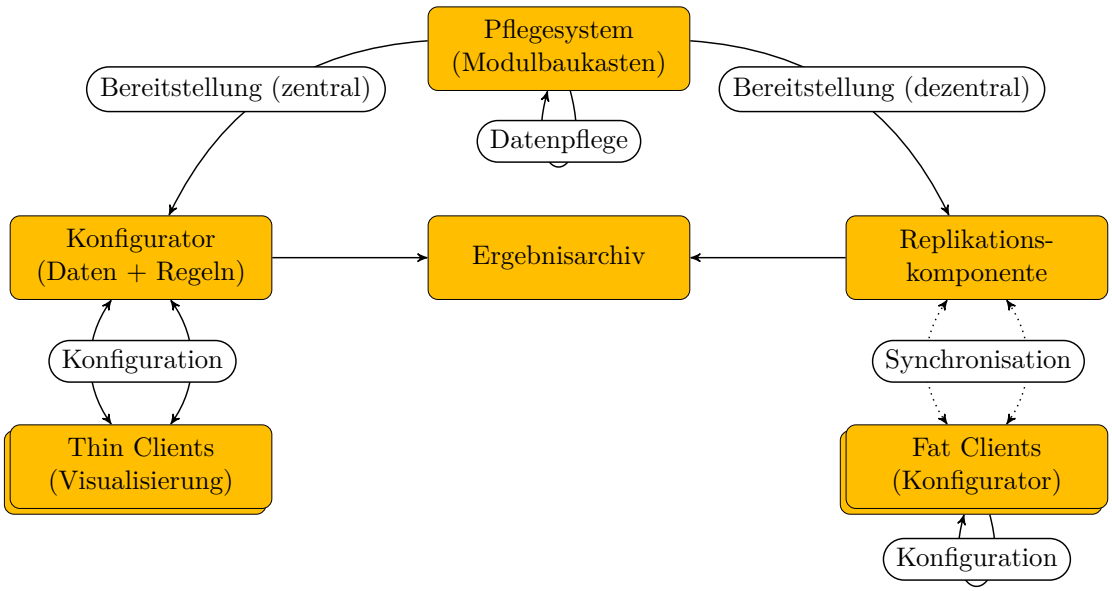


Abbildung 2.7: Zentrales und dezentrales Konfigurationsszenario

[2] `http://www.opensource.org`

Die Organisation der Datenpflege kann sowohl in einer zentralen Form als auch über verteilte Systeme gestaltet werden. Dabei steht immer die Konsistenz des Modulbaukastens im Vordergrund, deren Gewährleistung durch den Einsatz eines gemeinsamen Pflegesystems geringeren Aufwand erfordert. Die anschließende Datenbereitstellung dient der Freigabe eines geprüften Modellierungsstands. Dabei wird in der zentralen Variante der auf einem Anwendungsserver laufende Konfigurator mit den entsprechenden Daten und Regeln versorgt, aufgrund der dauerhaften Verbundenheit von Clients kann eine effiziente Aktualisierung und Wartung jener singuläre Anwendungs-Instanz erreicht werden. Die sogenannten *Thin Clients* visualisieren letztlich nur die Konfiguratorergebnisse, beispielsweise über einen Web-Browser, und benötigen hierfür keinerlei eigene Anwendungslogik. Auch die Überführung und zentrale Weiterverarbeitung von abgeschlossenen Produktzusammenstellungen ist aufgrund der dauerhaften Verbindung aus technischer Sicht ohne Verzögerungen möglich.

Dagegen besteht im dezentralen Architekturszenario keine dauerhafte Verbindung der Clients zu einem Server innerhalb eines gemeinsamen Datennetzes. Um auch im unverbundenen Zustand mit dem Konfigurator arbeiten zu können, ist es notwendig, sowohl die entsprechenden Daten und Regeln als auch die eigentliche Anwendungslogik auf diesen dann als *Fat Clients* bezeichneten Systemen vorzuhalten. Statt einer zentralen Anwendungs-Instanz existieren hier also eine Vielzahl funktional eigenständiger Ausprägungen des Produktkonfigurators. In diesem Zusammenhang übernimmt eine dedizierte Replikationskomponente die Aktualisierung und Synchronisierung von Clients bezüglich der Modelldaten und offline erzeugten bzw. geänderten Konfigurationen bei vorliegender Verbundenheit. Auf Details zu Herausforderungen und Lösungsansätzen hierzu wird ausführlich in [Gol06] eingegangen. Praktische Relevanz hat dieses Szenario im Außendienstbereich zur Planung und Konfiguration direkt beim Kunden, wenn trotz fortlaufender Verbesserung der mobilen Netz-Infrastruktur eine leistungsfähige permanente Datenverbindung nicht gegeben oder aufgrund unternehmensrelevanter Einschränkungen nicht gewünscht ist.

2.3.2.4 Persistierung

Wie bei anderen datenverarbeitenden Informationssystemen besteht auch in Produktkonfiguratoren für die Unterstützung von komplexen Prozessen (siehe Abschnitt 2.3.3) die Notwendigkeit zur dauerhaften Speicherung von Daten. Hierbei kann entweder das vom Betriebssystem bereitgestellte *Dateisystem* oder ein *Datenbankmanagementsystem (DBMS)* zum Einsatz kommen. Aufgrund der verschiedenen Aufgaben der einzelnen in Abbildung 2.7 dargestellten Komponenten und den damit verbundenen Anforderungen wird die Persistierungsebene in der Praxis typischerweise individuell für das Pflegesystem, den eigentlichen Konfigurator und die Komponente zur Sicherung von Konfigurationsergebnissen festgelegt. Dabei kann die jeweilige Entscheidung zwischen einem Datei- oder Datenbanksystem auf Basis der generellen Kriterien für Datenhaltungssysteme [HR01] getroffen werden. Beispielsweise bietet sich für ein zentrales Pflegesystem die Speicherung des Modulbaukastens in einer Datenbank an, um dessen Konsistenz trotz eines potentiellen bzw. gewünschten Mehrbenutzerbetriebs über den integrierten Transaktionsschutz gewährleisten zu können. Dagegen spielt für das Konfigurationssystem auf einem zentralen Applikationsserver die Performance der vorrangig lesenden Zugriffe die größte Rolle, weswegen der Funktionsumfang eines Dateisystems meist ausreichend ist. Hierfür müssen jedoch die Daten des Pflegesystems in geeignete Strukturen des Dateisystems überführt werden.

2.3.3 Prozesse

Die Merkmale der Prozess-Kategorie repräsentieren die notwendigen Abläufe zwischen den in Abbildung 2.7 skizzierten Komponenten eines Konfigurationssystems. Gemäß den Anforderungen und Wünschen der hierbei involvierten Fachbereiche werden initial die Produktdaten im Rahmen der Modellierung und Pflege aufbereitet, anschließend erfolgt durch deren Bereitstellung die Freigabe zur Konfiguration. Aufgrund neuer Kundenwünsche und Produktentwicklungen wird diese in **Abbildung 2.8** dargestellte Prozesskette typischerweise zyklisch durchlaufen. Die folgenden Abschnitte erläutern die einzelnen Schritte im Detail.

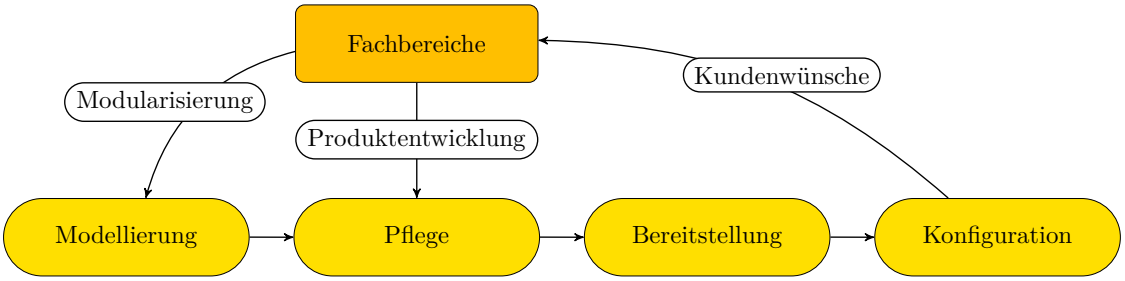


Abbildung 2.8: Übersicht zu Prozessen in Produktkonfiguratoren

2.3.3.1 Modellierung

Soll ein Produkt konfiguriert werden, sind geeignete Strukturen zur Abbildung und Prozessierung der Komponenten und Abhängigkeiten notwendig. Der Aufbau dieser auch als Modulbaukasten [Els03] bezeichneten Strukturen ist das Ziel der Modellierung, wobei die Modularisierung wie in Abschnitt 2.2.2.1 erläutert eine grundlegende Voraussetzung für die betreffenden Produkte darstellt. In diesem Zusammenhang spielt auch die Gleichteilestrategie zur Optimierung der gegenläufigen Bestrebungen von innerer und äußerer Varianz (siehe Abschnitt 2.2.2.1) eine wichtige Rolle [Pli12]. Die durchzuführenden Analysen und Abstraktionsaufgaben sind meist einem kleinen Expertenteam vorbehalten, welches jedoch die Erfahrungen und Anforderungen der einzelnen Fachbereiche beachten sollte.

2.3.3.2 Pflege

Nach Festlegung der Modellierungs-Strukturen können diese nun im Prozess der Pflege mit den Daten der zu konfigurierenden Produkte befüllt werden. Dies umfasst sowohl beschreibende Informationen zu den Komponenten wie Texte, Bilder und Preise als auch die Definition von Abhängigkeiten zwischen diesen (siehe Abschnitt 2.2.2.2), um die gewünschte Charakteristik des zusammengestellten Produkts zu gewährleisten. Bereits existierende Daten zu den Produkten, beispielsweise in einem ERP-System, lassen sich gegebenenfalls über Schnittstellen importieren. Die Abgrenzung des Pflege-Prozesses von der Modellierung kann letztlich mit jener Differenzierung von Sprachkonstrukten innerhalb der Structured Query Language (SQL) in die beiden Teile der Data Definition Language (DDL) und der Data Manipulation Language (DML) verglichen werden.

2.3.3.3 Bereitstellung

Basierend auf den gepflegten Baukastendaten ließe sich direkt die eigentliche Konfiguration durchführen. Allerdings wird typischerweise ein zusätzlicher Prozess zur Bereitstellung der Daten zwischengeschaltet, um einerseits den Einfluss der in Abbildung 2.7 dargestellten Szenarien auf die einzelnen Komponenten eines Produktkonfigurators zu minimieren und andererseits sicherheitsrelevante bzw. organisatorische Anforderungen erfüllen zu können. Außerdem bietet sich dadurch die Möglichkeit zur Transformation der Daten, um beispielsweise mit einem Wechsel der Persistierungsebene wie in Abschnitt 2.3.2.4 beschrieben der Zugriffscharakteristik des Konfigurations-Prozesses durch Vorübersetzung in laufzeitoptimierte Datenstrukturen Rechnung zu tragen.

2.3.3.4 Konfiguration

Für Anwender von Produktkonfiguratoren ist die Ausübung der Konfiguration der einzig sichtbare und damit entscheidende Prozess. Hierbei erfolgt die Zusammenstellung des gewünschten Produkts durch Bewertung von Fragen bzw. Merkmalen, wodurch die Aus- oder Abwahl einzelner Komponenten gemäß der gepflegten Abhängigkeiten (siehe Abschnitt 2.2.1) erfolgt. Die Möglichkeiten zur Steuerung dieses Verhaltens werden nachfolgend detailliert in Abschnitt 2.3.5 sowie Abschnitt 2.3.6 vorgestellt. Zusätzlich stehen üblicherweise Funktionen wie Laden, Speichern und Drucken bereit, um die Weiterverarbeitung der Konfigurationsergebnisse zu gewährleisten. Darüber hinaus kann auch die Protokollierung und Auswertung der einzelnen Konfigurationsprozesse wertvolle Hinweise zur Akzeptanz des Produktmodells liefern und lässt sich als eine Form von Kundenwünschen bei der iterativen Datenpflege integrieren.

2.3.4 Daten

In der Daten-Kategorie stehen deren Merkmale für unterschiedlichste Arten von Informationen, welche entweder die Grundlage oder das Ergebnis der zuvor in Abschnitt 2.3.3 erläuterten Prozesse bilden können. Entsprechend der in **Abbildung 2.9** dargestellten Prozesskette findet nachfolgend die Charakterisierung von Modelldaten, Oberflächendaten, Konfigurationsdaten, Verkehrsdaten, Administrationsdaten und Integrationsdaten statt.

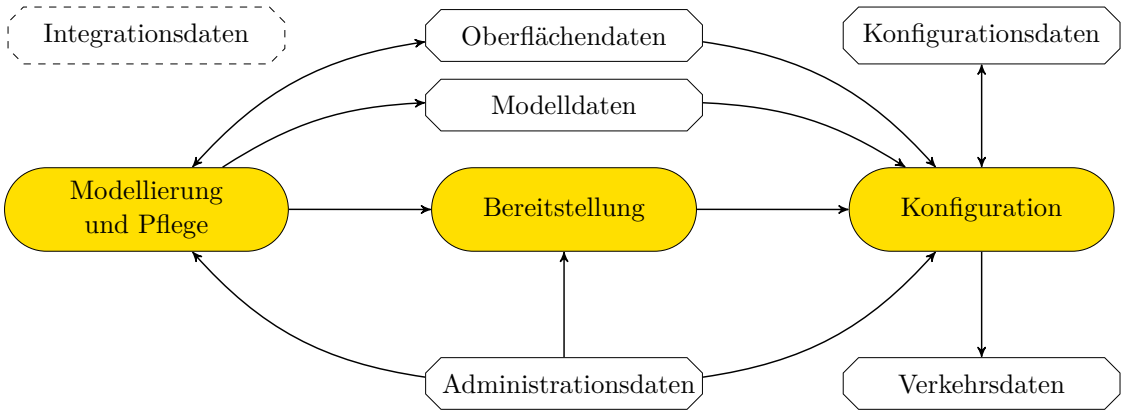


Abbildung 2.9: Beziehungen zwischen Daten und Prozessen in Produktkonfiguratoren

2.3.4.1 Modelldaten

Unter den Modelldaten werden jegliche Informationen zusammengeführt, die den Aufbau und die Eigenschaften des zu konfigurierenden Produkts beschreiben und im Rahmen der Modellierung bzw. Datenpflege zu bearbeiten sind. Diese Datenmenge umfasst insbesondere alle Produktkomponenten mit ihren zur Prozessierung und Visualisierung geeigneten Eigenschaften wie Texte, Preise und Bilder, jegliche Regeln und Abhängigkeitsdefinitionen zwischen den einzelnen Komponenten sowie Selektionsmerkmale zur Steuerung des Konfigurationsvorgangs. Dadurch genügen die Modelldaten den in Abschnitt 2.2.2 formulierten Voraussetzungen und stellen somit das insgesamt verfügbare Produktwissen dar.

2.3.4.2 Oberflächendaten

Dem Merkmal Oberflächendaten werden alle Informationen zugeordnet, welche die nach außen sichtbare Darstellung und Präsentation des Produktkonfigurators produktunabhängig festlegen. Hierzu zählen beispielsweise Beschriftungen für Bedienelemente wie Menüs und Buttons, Stylesheet-Definitionen mit Farbcodierungen und Schriftarten, anbieterbezogene Elemente zur Umsetzung des Corporate Designs sowie Layout-Spezifikationen. Häufig werden jene Daten während der Implementierung eines Konfigurationssystems mit standardisierten Werten hinterlegt und erst parallel zur Datenpflege und Modellierung spezifisch für den Anbieter je nach Vorgabe ausgeprägt.

2.3.4.3 Konfigurationsdaten

Die Konfigurationsdaten repräsentieren jene Daten, die im Rahmen eines vom Nutzer durchgeführten Konfigurationsprozesses entstehen. Dabei sind einzig die Informationen zur Charakterisierung des Produkts von Bedeutung, wie beispielsweise die Bewertung von Selektionskriterien oder die Menge der ausgewählten Optionalkomponenten. Letztendlich muss basierend auf diesen Daten die Wiederherstellung eines gespeicherten Konfigurationszustands zur weiteren Verarbeitung möglich sein.

2.3.4.4 Verkehrsdaten

Als Verkehrsdaten lassen sich alle Daten klassifizieren, die während eines Konfigurationsprozesses zwischen Nutzer und Konfigurator ausgetauscht werden und nicht zu den Konfigurationsdaten zählen. Typische Beispiele hierfür sind Kontakt- und Adressdaten zur Angebotserstellung sowie auf technischer Ebene die IP-Adresse und weitere Systeminformationen über den vom Nutzer verwendeten Client. Aufgrund der Charakterisierung als personenrelevante Informationen sind für Verkehrsdaten die Richtlinien des Datenschutzes sicherzustellen, sofern der Konfigurator im Internet bereitgestellt wird.

2.3.4.5 Administrationsdaten

Das Merkmal der Administrationsdaten umfasst Informationen, welche für den Betrieb eines Konfigurators aus organisatorischer und technischer Sicht nötig sind. Dazu gehören beispielsweise Nutzerberechtigungen, Rollenzuordnungen sowie Zugangsdaten für die involvierten Systeme.

2.3.4.6 Integrationsdaten

Dieses Merkmal nimmt innerhalb der Daten-Kategorie eine Sonderrolle durch die Differenzierung in zwei Gruppen ein. Einerseits bezeichnen die *globalen* Integrationsdaten jene Informationen, welche durch die Kopplung von fremden Systemen mit einem Produktkonfigurator beidseitig ausgetauscht werden können. Beispielsweise lassen sich Produktstrukturen als Teil der Modelldaten aus einem ERP-System als Basis für die Datenpflege importieren oder Konfigurationsdaten zur Auftragsbearbeitung exportieren. Die Informationen zur Realisierung der dafür notwendigen Verbindung von Systemen werden dagegen den *lokalen* Integrationsdaten zugeordnet. Beispiele hierfür sind technische Anforderungen und Schnittstellen-Spezifikationen einer zu koppelnden externen Anwendung.

2.3.5 Modellbildung

Aufgrund der enormen Bedeutung der Modelldaten für ein Konfigurationssystem soll der in Abschnitt 2.3.3.1 vorgestellte zugehörige Prozess durch die Kategorie der Modellbildung erneut aufgegriffen und detailliert betrachtet werden. Dafür werden nachfolgend die Möglichkeiten der Beeinflussung durch den Rekursionsgrad, die Unterstützung einer Produktwissensevolution, den Gestaltungsspielraum und die Art sowie den Zeitpunkt von Abhängigkeitsprüfungen analysiert.

2.3.5.1 Rekursionsgrad

Die Konfiguration eines Produkts entspricht gemäß Definition 2.1 und Abschnitt 2.2.1 der Zusammenstellung seiner Komponenten. Der Rekursionsgrad gibt dabei die Tiefe der Komponentenbildung als Ergebnis einer *einstufigen* oder *mehrstufigen* Modellierung an, welche sich direkt auf den Individualisierungsgrad und die Komplexität des Datenmodells auswirkt. Dies lässt sich an der Konfiguration eines Autos veranschaulichen. Im einstufigen praxisrelevanten Fall ist nur die Auswahl der Grundkomponenten (z.B. Motorvarianten) möglich. Im mehrstufigen Szenario könnte stattdessen der Motor hinsichtlich seiner technologischen Merkmale (z.B. Zylinderzahl, Leistung, Drehmoment) individuell spezifiziert werden. Die rekursive Fortsetzung dieses Prinzips ist allerdings praktisch begrenzt durch die Bestrebung zur Minimierung der Fertigungsvarianz (siehe Abschnitt 2.2.2.1) sowie das bei der Zielgruppe voraussetzbare Wissen zum Produktaufbau (siehe Abschnitt 2.3.1.2).

2.3.5.2 Produktwissensevolution

Bedingt durch unternehmensinterne Aktivitäten sowie externe Einflussnahme des Marktes unterliegen Produkte in ihrem Produktlebenszyklus einer stetigen Entwicklung (siehe auch Abbildung 2.8). Gemäß der in Abschnitt 2.2.3.3 formulierten Anforderung zum flexiblen Konfigurationswissen sollten derartige Änderungen in einem Konfigurationssystem bzw. Pflegesystem erfasst und verwaltet werden können, wobei diese einer von drei Kategorien angehören. Im Fall der *teilweisen* Unterstützung sind die Anpassungen auf gewisse Bereiche des Datenmodells beschränkt, beispielsweise auf die Beschreibung von Komponenten ohne Veränderung der eigentlichen Produktstruktur. Dagegen können bei der *vollständigen* Produktwissensevolution jegliche Entscheidungen der initialen Modellierung zu einem späteren Zeitpunkt geändert werden mit gegebenenfalls weitreichenden

Auswirkungen auf die Abhängigkeitsdefinitionen. Daneben bieten Konfigurationssysteme aktive Unterstützung beim Wechsel zwischen den einzelnen Produktentwicklungsstufen, wenn das Datenmodell *versioniert* verwaltet und ein Änderungsprotokoll mitgeführt wird.

2.3.5.3 Gestaltungsspielraum

Während durch den Rekursionsgrad die Struktur von Produkten und Komponenten primär aus Modellierungssicht betrachtet wird, bildet die daraus resultierende auf Anwenderseite verfügbare Vielfalt zur Produktindividualisierung die Grundlage für das Merkmal des Gestaltungsspielraums. Diesbezüglich können abhängig von der Begriffsdefinition für Module und Komponenten gemäß [Sch06] die folgenden vier Klassen unterschieden werden. Die sogenannte *Mixkonfiguration* ist gekennzeichnet durch die Kombination beliebiger atomarer Komponenten (siehe Definition 2.1), zwischen denen keine oder nur wenige Abhängigkeiten existieren. Ein typisches Beispiel ist der Müsli-Konfigurator[3]. Dagegen spricht man von einer *Modulkonfiguration*, wenn die als Module bezeichneten verfügbaren Komponenten einen strukturierten Aufbau besitzen, der sich prinzipiell wiederum konfigurieren ließe, und deutlich komplexere Abhängigkeitsbeziehungen zwischen diesen existieren. Der Auto-Konfigurator von Porsche[4] sei hier stellvertretend genannt. Ist andererseits die Grundform des Produkts bereits festgelegt und nur an definierten Ansatzpunkten veränderbar, handelt es sich um eine *Anpassungskonfiguration*. Dieses Prinzip wird unter anderem beim Jeans-Konfigurator[5] deutlich. Schließlich kann im Rahmen der *Designkonfiguration* sogar die Grundform des Produkts sowie dessen inhaltliche Spezifikation individuell im Sinne einer Konstruktion gestaltet werden, womit bereits nach der Abgrenzung und Begriffsprägung in Abschnitt 2.2.1 der Produktentstehungsprozess beeinflusst wird.

2.3.5.4 Art der Prüfung

Wie in Abschnitt 2.2.2.2 erläutert, bildet die formale Beschreibung von Abhängigkeiten zwischen den durch die Modularisierung entstandenen Produktkomponenten eine Wissensbasis, welche die zentrale technische Komponente zur Gewährleistung einer konsistenten Produktkonfiguration darstellt [BAKF04]. Für die Spezifikation derartiger Abhängigkeitsprüfungen werden hier Regeln, Entscheidungsbäume und Entscheidungstabellen als die wichtigsten in der Literatur genannten und praxisrelevantesten Ausprägungen kurz charakterisiert. Zu weiterführenden Details sowie Informationen über alternative Techniken wie beispielsweise den fallbasierten oder objektbasierten Systemen sei auf die Arbeiten von [Lec06, Sch06] und [Wüp00] verwiesen.

Die Wissensrepräsentation mit Hilfe von *Regeln* ist aufgrund ihrer Nähe zum menschlichen Denken die am häufigsten verwendete Technik. Dabei lassen sich typische Handlungsanweisungen als Konditionalsätze gemäß dem Schema „wenn (Bedingung), dann (Aktion)“ formalisieren. Bezüglich der logischen Interpretierbarkeit entsprechen der Bedingungs- und Aktionsteil Formeln der Prädikatenlogik erster Stufe. Dadurch sind sie leicht automatisch prozessierbar, allerdings ist die übersichtliche Darstellung komplexer Regelwerke im Rahmen der Modellierung und Pflege effektiv nicht möglich. Hierfür ist das Konstrukt der *Entscheidungsbäume* besser geeignet, da sie eine Menge inhaltlich zusammengehörender

[3] `http://www.mymuesli.com`
[4] `http://www.porsche.com/germany/carconfiguratorgeneral`
[5] `http://www.smart-jeans.com`

Regeln in kompakter Form repräsentieren. Jede Ebene eines solchen Entscheidungsbaums stellt die weiteren im Konfigurationsprozess verfügbaren Auswahlmöglichkeiten dar, wenn zuvor der übergeordnete Vater-Knoten selektiert wurde. Damit wird jedoch implizit die Auswahl-Reihenfolge für den Nutzer definiert, ein wahlfreier Konfigurationsablauf ist nicht möglich. Dieser Einschränkung unterliegen *Entscheidungstabellen* trotz einer ebenfalls sehr guten Visualisierung von Abhängigkeiten nicht. Dabei stellen die Dimensionen einer Matrix die miteinander korrelierenden Merkmale inklusive ihrer Werte und Eigenschaften dar. Typischerweise symbolisiert eine konkrete Wertbelegung (z.B. ein Kreuz) die Verträglichkeit aller in diesem Punkt aufeinander treffenden Dimensions-Ausprägungen.

In **Abbildung 2.10** ist stellvertretend am Beispiel einer imaginären Abhängigkeit zwischen den Merkmalen Fahrzeugtyp („Typ“) und Motorisierung („Motor“) für einen Automobil-Konfigurator die zugehörige Spezifikation eines Regelsatzes, eines Entscheidungsbaums und einer Entscheidungstabelle verdeutlicht.

```
Wenn Typ=Cabrio, dann Motor=Benzin oder Motor=Elektro
Wenn Typ=Kombi, dann Motor=Diesel oder Motor=Benzin oder Motor=Elektro
Wenn Typ=Coupe, dann Motor=Benzin oder Motor=Diesel
```

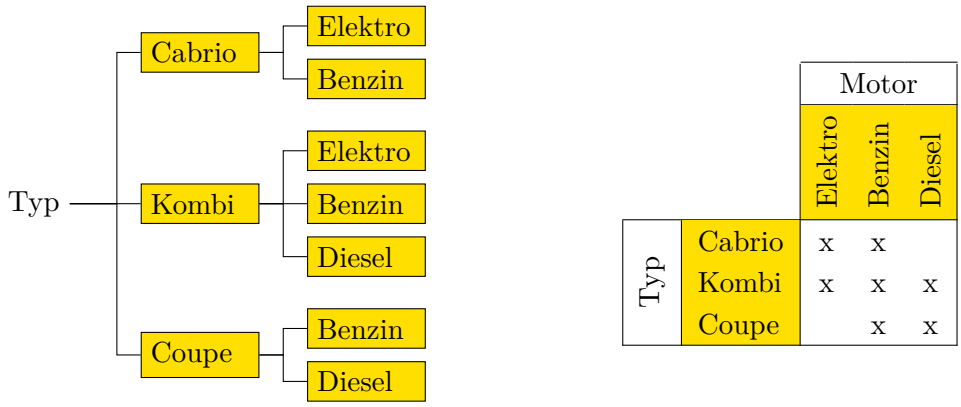


Abbildung 2.10: Beispiele für Abhängigkeitsprüfungen in Produktkonfiguratoren

2.3.5.5 Zeitpunkt der Prüfung

Neben den vorab beschriebenen Varianten zur Spezifikation von Abhängigkeiten lassen sich Produktkonfiguratoren auch hinsichtlich des Zeitpunkts für die Prüfung dieses modellierten Beziehungswissens unterscheiden. Aufgrund der Tatsache, dass gemäß Abschnitt 2.2.1 zumindest nach Abschluss des Konfigurationsprozesses eine konsistente Zusammenstellung gewährleistet sein muss, lassen sich drei Strategien zur Durchführung der Abhängigkeitsprüfung charakterisieren. Bei der *laufzeitbezogenen* Methode stößt jede Veränderung des Konfigurationszustands den Prüfvorgang erneut an. Dadurch bekommt der Nutzer zu seinen Aktivitäten beeinflusst von der in Abschnitt 2.3.6.4 diskutierten Optionsdarstellung eine direkte Rückmeldung. Außerdem wird sichergestellt, dass sich eine Konfiguration nach Auflösung potentieller Konfliktsituationen zu jedem Zeitpunkt in einem gültigen Zustand befindet. Allerdings erfordert diese Strategie eine erhöhte Rechenleistung zur Verarbeitung aller relevanten Abhängigkeiten und kann mitunter zu zeitlichen Verzögerungen in

der Nutzerinteraktion führen. Diesem Problem kann insbesondere im Umfeld geschulter Nutzer durch die sogenannte *intervallbezogene* Technik begegnet werden, bei der nur an bestimmten Punkten im Konfigurationsverlauf die Prüfung aller Abhängigkeitsdefinitionen vorgenommen wird. Zwischen diesen Punkten, welche beispielsweise entsprechend einer inhaltlichen Gruppierung von Produktkomponenten definiert sein können, erfolgt die Bewertung und Auswahl durch den Nutzer ohne Überwachung seitens des Konfigurators und damit ohne zusätzliche Verzögerungen. Bei konsequenter Fortsetzung dieses Prinzips erhält man letztlich die *ergebnisbezogene* Variante, in welcher erst nach Abschluss des Konfigurationsprozesses eine Prüfung des gesamten Beziehungswissens stattfindet. Falls hierbei jedoch ungültige Kombinationen erkannt werden, muss der Nutzer im Extremfall die komplette Konfiguration anpassen. Diese Art der Verarbeitung hat ihre Wurzeln in der Epoche der stapelverarbeitenden Systeme und spielt in den heutigen interaktiven Konfigurationssystemen keine bedeutende Rolle mehr.

2.3.6 Präsentation

Nachdem mit Abschnitt 2.3.5 die Abläufe der Modellierung und Datenpflege genauer analysiert wurden, steht abschließend der eigentliche Konfigurationsprozess im Fokus der Betrachtungen. Dazu umfasst die Präsentations-Kategorie jene Merkmale eines Produktkonfigurators, deren Ausprägungen das direkt sichtbare Verhalten und Erscheinungsbild gegenüber der in Abschnitt 2.3.1.2 beschriebenen Zielgruppe beeinflussen. Nachfolgend werden hierfür die Visualisierungsform, der Startzustand, das Konfigurationsprinzip, die Optionsdarstellung und -kategorisierung sowie die Vorschlagsbildung näher erläutert.

2.3.6.1 Visualisierungsform

Für die Visualisierung insbesondere der Modell- und Oberflächendaten (siehe Abschnitt 2.3.4) lassen sich gemäß den Ausführungen in [Lec06] drei Varianten abgrenzen. Die ausschließliche Verwendung von *Text* als Medium zur Informationsvermittlung ermöglicht einen zielgerichteten und effizienten Konfigurationsprozess. Allerdings kommt aufgrund der fehlenden grafischen Elemente für Produkterläuterungen und die Nutzerführung diese Art eines Konfigurations-Frontend typischerweise nur bei internen Mitarbeitern oder Nutzern mit entsprechendem Expertenwissen zum Einsatz. Um auch ungeschulten Endanwendern eine intuitive und ansprechende Konfiguration zu gewährleisten, gehört heutzutage die Anzeige *statischer Grafik* im Sinne von vordefinierten Bildern, Videos oder Flash-Medien für die Produktkomponenten zum üblichen Funktionsumfang von Produktkonfiguratoren. Darüber hinaus wächst seit den letzten Jahren die Zahl der Konfigurationssysteme, welche eine Visualisierung von *dynamischer Grafik* unterstützen. Hierbei wird dem Nutzer ein zur Laufzeit erzeugtes und zum aktuellen Konfigurationszustand passendes Bild oder interaktives 3D-Modell des zusammengestellten Produkts dargestellt, um einen realistischeren Eindruck zu vermitteln. Im Fall des Porsche-Konfigurators betrifft dies beispielsweise die Anzeige von Farben, Felgen sowie einige Optionen der Innen- und Außenausstattung.

2.3.6.2 Startzustand

Das Merkmal des Startzustands charakterisiert Produktkonfiguratoren bezüglich der Vorauswahl von Produktkomponenten beim initialen Aufruf, welche direkten Einfluss auf

die Selektionsstrategie des Nutzers hat. Hierbei können nach [Pol08, Sch06] grundsätzlich zwei Alternativen unterschieden werden. Die sogenannte *Neukonfiguration* oder auch *Bottom-Up-Methode* beginnt den Konfigurationsprozess ausgehend von einer leeren Produktkomponentenmenge, es findet also keinerlei Vorbelegung statt. Um eine konsistente Zusammenstellung zu erhalten, muss mindestens die Auswahl der Basiskomponenten durch den Nutzer erfolgen. Dagegen startet man im Fall der *Basiskonfiguration* oder auch *Top-Down-Methode* mit einer Menge an vorausgewählten und durch den Anbieter bestimmten Produktkomponenten (siehe Abschnitt 2.3.6.6), welche bereits eine gültige Konfiguration ergeben. Der Nutzer hat im Verlauf des Konfigurationsprozesses die Möglichkeit, die bestehende Zusammenstellung durch An- und Abwahl von Produktkomponenten zu verändern, wobei deren Gültigkeit vom Zeitpunkt der Prüfung gemäß Abschnitt 2.3.5.5 abhängt.

2.3.6.3 Konfigurationsprinzip

Nach vorheriger Analyse des Startzustands werden nun im aktuellen Merkmal die beiden verschiedenen Prinzipien des Konfigurationsprozesses betrachtet. Die am häufigsten in der Praxis anzutreffende Variante ist die sogenannte *optionsbasierte Konfiguration*, in der die An- und Abwahl der einzelnen Produktkomponenten bewusst durch den Nutzer im Rahmen eines interaktiven Vorgangs gesteuert wird. Hierbei lässt sich noch differenzieren, ob für den Konfigurationsprozess ein bestimmtes Ablaufmuster im Rahmen der Modellbildung festgelegt wurde (siehe Abschnitt 2.3.5) oder stattdessen der Nutzer die Bewertung der einzelnen Optionen in beliebiger Reihenfolge durchführen kann. Ein generelles Beispiel stellt die Auswahl eines Motors im Fahrzeugkonfigurator anhand der verfügbaren Leistungswerte dar. Soll jedoch die Zusammenstellung anhand von Anforderungen erfolgen, die sich zwar für das gesamte Produkt formulieren lassen, aber nicht notwendigerweise einer Komponente direkt zuordenbar sind, ist hierfür die *zielgrößenbasierte Konfiguration* anzuwenden. In dieser eher deskriptiven Arbeitsweise definiert der Nutzer über geeignete Kriterien seine Anforderungen an das zu konfigurierende Produkt. Anschließend versucht das Konfigurationssystem automatisch, eine gültige Zusammenstellung der Produktkomponenten abzuleiten, welche die Vorgaben erfüllen. Im Kontext der Fahrzeugkonfiguration könnte beispielsweise der Wunsch nach dem verbrauchärmsten Auto mit den geringsten Anschaffungskosten eine solche Anforderung repräsentieren.

2.3.6.4 Optionsdarstellung

Die Anwendung einer laufzeit- oder intervallbezogenen Abhängigkeitsprüfung (siehe Abschnitt 2.3.5.5) bietet neben der Sicherstellung konsistenter Konfigurationszustände zeitnah zu den nutzerspezifischen Selektionen auch die Berücksichtigung der resultierenden Variantenvielfalt für die Visualisierung. Bei der *eingeschränkten Optionsdarstellung* werden ausgehend von den getroffenen Entscheidungen des Nutzers nur noch die gültigen Optionen und Produktkomponenten im weiteren Konfigurationsprozess angezeigt oder als verfügbar gekennzeichnet, d.h. deren Auswahl ist ohne Konflikt zum gesamten Beziehungswissen möglich. Die dadurch gewonnene Übersichtlichkeit ist jedoch mit einer Reduktion der Transparenz verbunden, weil dem Nutzer potentielle Alternativen vorenthalten werden. Beispielsweise wäre in Abbildung 2.10 die Auswahl eines Elektromotors nach Festlegung auf den Fahrzeugtyp Coupe nicht mehr möglich. Dagegen werden bei der *vollständigen Optionsdarstellung* immer alle Produktkomponenten unabhängig vom Kon-

figurationszustand angeboten. Falls der Nutzer eine Auswahl trifft, die zur Verletzung von Abhängigkeiten führen würde, bekommt er typischerweise in einem Konfliktdialog einen Hinweis mit Möglichkeiten zur Auflösung des Konflikts. Beim genannten Beispiel in Abbildung 2.10 bliebe der Elektromotor auch nach Auswahl eines Coupes verfügbar, führt aber bei tatsächlicher Selektion zur Abwahl des aktuellen Fahrzeugtyps und zur Neubewertung durch den Nutzer, wobei dann nur noch Cabrio oder Kombi konfliktfrei wählbar sind.

2.3.6.5 Optionskategorisierung

Mit dem Merkmal der Optionskategorisierung wird die Anordnung der zur Auswahl stehenden Produktkomponenten charakterisiert. Dabei lässt sich eine eher *unstrukturierte Darstellung* von einer *inhaltlichen Gruppierung* unterscheiden. Beispielsweise existieren in einem Fahrzeugkonfigurator üblicherweise Gruppen für das Exterieur, die Sitze und die Innenausstattung. Hierbei zeigen Studien, dass die inhaltliche Kategorisierung das Entscheidungsverhalten von Nutzern mit wenig Produktwissen, d.h. von typischen Endanwendern, beeinflusst und ihnen als Orientierungshilfe dient [Pol08]. Dagegen benötigen geschulte Nutzer laut jener Studie derartige Unterstützung nicht, da ihnen das Expertenwissen entsprechende Sicherheit im Umgang mit der Produktvielfalt gibt. Somit stellt die Optionskategorisierung neben der in Abschnitt 2.3.6.1 erläuterten Visualisierungsform ein weiteres Kriterium für die zielgruppenspezifische Ausprägungen eines Konfigurators dar.

2.3.6.6 Vorschlagsbildung

Neben der Gruppierung von Produktkomponenten bietet auch die Generierung und Anzeige von Vorschlägen eine effektive Möglichkeit zur Beeinflussung des in Abschnitt 2.3.6.5 angesprochenen Entscheidungsverhaltens. Im vorliegenden Merkmal werden dafür basierend auf [Pol08] zwei Modellierungsansätze unterschieden. Einerseits lassen sich sogenannte *Defaults* definieren, welche automatisch zur statischen oder abhängigkeitsbedingten Vorauswahl von Optionen bzw. Produktkomponenten durch den Konfigurator führen. Anwendung findet diese Methode beispielsweise bei Konfiguratoren, deren Startzustand dem in Abschnitt 2.3.6.2 erläuterten Prinzip der Basiskonfiguration entspricht. Andererseits stellen *Empfehlungen* unterstützende Hinweise zur geeigneten Fortsetzung des Zusammenstellungsprozesses dar. Diese können sowohl bedingungslos als auch abhängig vom aktuellen Konfigurationszustand formuliert werden. Weiterhin sind in diesem Kontext unpersonalisierte von personengebundenen Empfehlungen zu unterscheiden [Lec06], wobei kundenspezifische Daten beispielsweise aus Cookies oder auch initial angeforderten Anmelde-Informationen gewonnen werden können.

2.4 Integratives Grundmodell

Basierend auf den Erkenntnissen der bisherigen Definitionen, Anforderungsanalyse und Klassifikation wird nachstehend das Konzept eines integrativen Grundmodells für Produktkonfiguratoren vorgestellt. In diesem Zusammenhang ist der Begriff des Grundmodells als Entwurfsmuster im Sinne eines abstrakten Referenzmodells für Konfigurationssysteme zu verstehen. Darüber hinaus erlaubt die Gestaltung als offenes System die Integration in eine bestehende Anwendungslandschaft.

Die Gesamtarchitektur des integrativen Grundmodells für Produktkonfiguratoren ist schematisch in **Abbildung 2.11** dargestellt. Darauf aufbauend findet dessen funktionale Beschreibung über drei Schwerpunkte statt. Im ersten Schritt werden die Aufgaben der einzelnen Module und Komponenten in Abschnitt 2.4.1 näher erläutert. Anschließend erfolgt in Abschnitt 2.4.2 die Analyse der Abhängigkeiten zwischen den Modulen aus Sicht des Kontrollflusses, bevor auch der Datenfluss in Abschnitt 2.4.3 betrachtet wird. Weiterführende Details, wie beispielsweise zu Synergieeffekten oder einer Anforderungsvalidierung des integrativen Grundmodells, sind in [Kru10] zu finden.

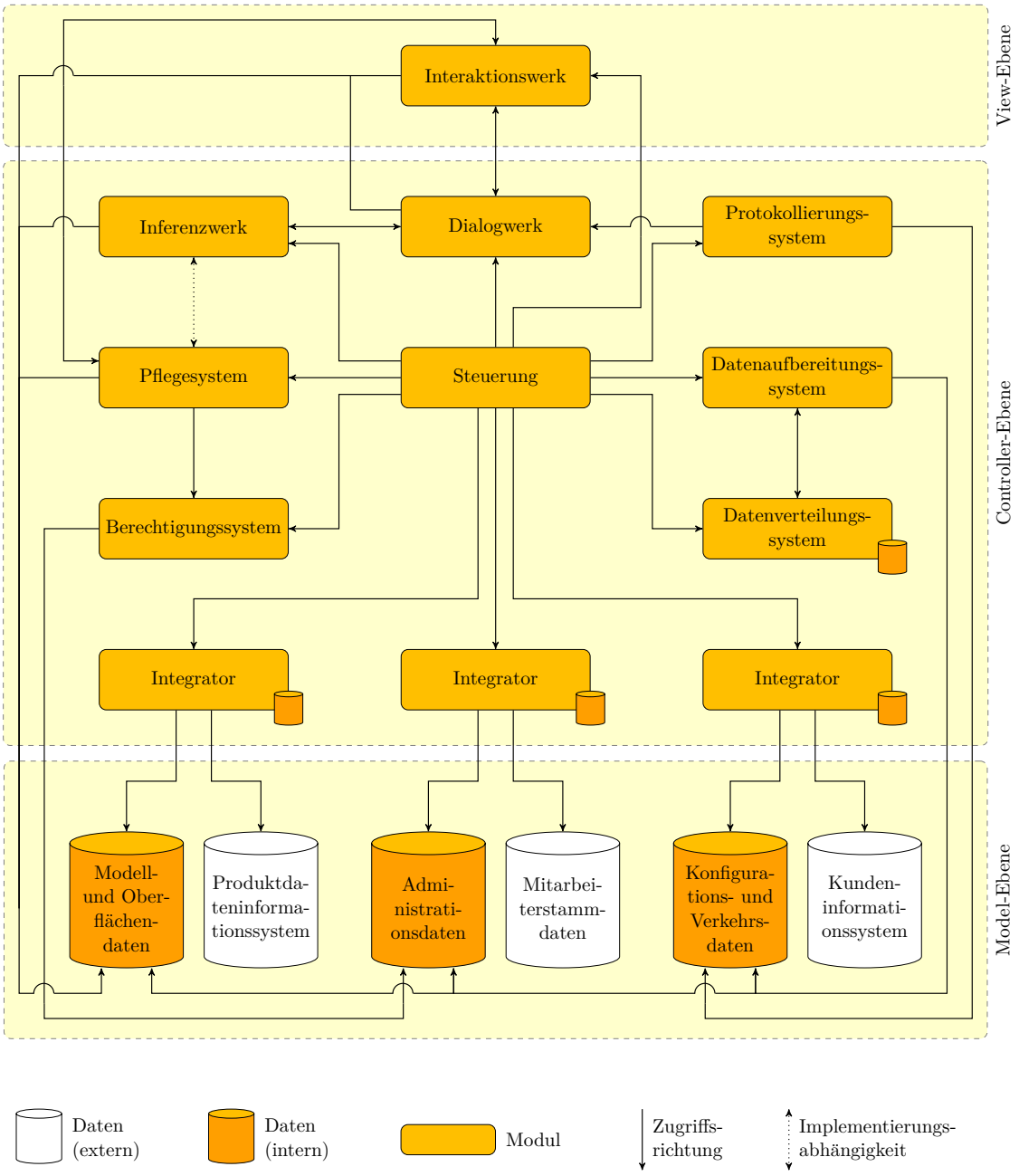


Abbildung 2.11: Integratives Grundmodell für Produktkonfiguratoren nach [Kru10]

2.4.1 Modulbeschreibung

Die Verwendung des Architekturmusters Model-View-Controller (MVC), dessen Begrifflichkeiten und Konzept [Ree79, Ree03] bereits vor über 30 Jahren entwickelt wurden, ermöglicht einen modularen und flexiblen Aufbau des Grundmodells. Dabei lassen sich zur Strukturierung die drei Ebenen Datenmodell (Model), Präsentation (View) und Programmsteuerung (Controller) unterscheiden. Im Kontext der Produktkonfiguratoren ist dieses Prinzip nicht nur für die Entwicklung der einzelnen Komponenten einsetzbar, sondern charakterisiert auch deren in Abbildung 2.7 dargestelltes Zusammenspiel.

2.4.1.1 Model-Ebene

Unabhängig von dem in Abschnitt 2.3.1.4 erläuterten Integrationsgrad, gehören der Model-Ebene die sogenannten *internen* Persistierungselemente an, welche gemäß den Ausführungen in Abschnitt 2.3.2.4 als Dateien oder Datenbanken ausgeprägt sein können. Diese sind verantwortlich für die Speicherung der verschiedenen in Abschnitt 2.3.4 klassifizierten Daten, wie beispielsweise den Modell- und Oberflächendaten, und stehen in direkter Verbindung mit der Mehrzahl der Module. Mit wachsender Einbettung eines Produktkonfigurators in die existierende Systemumgebung ist die Anbindung *externer* Persistierungselemente sinnvoll. Diese zählen zwar nicht unmittelbar zum Grundmodell, enthalten aber zum Aufbau der internen Daten relevante Informationen, beispielsweise beschreibende Texte und Bilder der konfigurierbaren Produktkomponenten. Für den Zugriff auf jene Anwendungen spielen die sogenannten *intern gekoppelten* Persistierungselemente eine Rolle, welche die in Abschnitt 2.3.4.6 klassifizierten Integrationsdaten verwalten und direkt an das Integrator-Modul (siehe Abschnitt 2.4.1.3) gebunden sind.

2.4.1.2 View-Ebene

Die Präsentationsschicht enthält im Grundmodell mit dem Interaktionswerk nur ein einziges Modul. Dessen Aufgabe besteht darin, sowohl das Erscheinungsbild des Konfigurators anhand von Oberflächenbeschreibungen und Style-Definitionen (siehe Abschnitt 2.3.4.2) zu generieren, als auch die Nutzereingaben entgegenzunehmen und deren weitere Verarbeitung zu gewährleisten. Bei diesem Prozess kommt es typischerweise zu Transformationen, beispielsweise um die Auswahl einer Komponente durch den Anwender in die für nachfolgende Module verständliche Syntax umzuwandeln. Letztendlich bildet das Interaktionswerk also die Schnittstelle zwischen dem Produktkonfigurator und seinen Benutzern.

2.4.1.3 Controller-Ebene

Gemäß der Darstellung in Abbildung 2.11 sind bis auf das Interaktionswerk in der View-Ebene alle Module des Grundmodells der Controller-Ebene zugeordnet, welche in diesem Abschnitt näher erläutert werden.

Steuerung: Für die Verwaltung aller im Grundmodell existierenden Bestandteile ist eine zentrale Einheit notwendig, welche sich im Modul Steuerung manifestiert. Hier liegen Informationen über die Ansteuerung, Kenndaten und Funktionsfähigkeit aller registrierten Module vor, d.h. die Steuerung übernimmt die Rolle einer zentralen Bus-Komponente im Sinne einer Enterprise Application Integration (EAI).

Berechtigungssystem: Um die autorisierte Nutzung des Produktkonfigurators gewährleisten zu können, ist bei allen Zugriffen das Modul Berechtigungssystem involviert. Dieses dient einerseits zur Authentifizierung eines Nutzers mit seinen Anmeldedaten und andererseits zur Ermittlung der ihm zugeordneten Rechte bezüglich des Aufrufs einer bestimmten Modulfunktion. Neben der Hinterlegung von Benutzergruppen und Zugriffsrechten im Persistierungselement für Administrationsdaten lassen sich auch externe Anwendungen integrieren, wie beispielsweise ein unternehmensweit verwendetes Lightweight Directory Access Protocol (LDAP) als Nutzerverwaltung.

Pflegesystem: Das Modul Pflegesystem ermöglicht die Durchführung der grundlegenden in Abschnitt 2.3.3 erläuterten Prozesse zur Modellierung und Pflege. Hierbei relevante Modell- und Oberflächendaten (siehe Abschnitt 2.3.4) werden im gleichnamigen Persistierungselement gespeichert und sollten üblicherweise nur von Anwendern mit entsprechender Fachkenntnis verändert werden. Eine hierzu notwendige Prüfung autorisierter Nutzerzugriffe erfolgt wie beschrieben im Modul Berechtigungssystem.

Inferenzwerk: Die Prüfung aller während des Konfigurationsprozesses auftretenden und in Abschnitt 2.3.5.4 klassifizierten Abhängigkeiten ist die Aufgabe des Moduls Inferenzwerk. Gemäß dessen Herleitung vom lateinischen Wort *infero* wird die Menge der notwendigen Konsequenzen gefolgert, um für den aktuellen Konfigurationszustand eine Widerspruchsfreiheit bezüglich der definierten Wissensbasis zu erreichen. Dies geschieht in eigenständiger Art und Weise ohne Eingriff seitens der Anwender.

Dialogwerk: Während das Interaktionswerk in der View-Ebene die Benutzeroberfläche eines Konfigurators generiert, stellt das Modul Dialogwerk mit den ereignisbezogenen Texten eine Teilmenge der dafür notwendigen Daten bereit. Primär sind dies zum aktuellen Konfigurationszustand passende Fehlermeldungen oder Hinweise beispielsweise nach Auswahl einer Produktkomponente. Dabei beeinflussen die in Abschnitt 2.3.6 vorgestellten Kriterien die tatsächliche Ausprägung des Moduls.

Protokollierungssystem: Mit Hilfe des eigenständig arbeitenden Moduls Protokollierungssystem lassen sich alle Informationen aufzeichnen, die aufgrund der Interaktion zwischen dem Produktkonfigurator und einem Nutzer entstehen. Gemäß der Charakterisierung in Abschnitt 2.3.4 betrifft dies sowohl die Konfigurationsdaten als auch die Verkehrsdaten, für deren Speicherung das zugehörige Persistierungselement verantwortlich ist.

Datenaufbereitungssystem: Zur Unterstützung eines dezentralen Architekturszenarios mit Fat Clients (siehe Abschnitt 2.3.2.3) wird das Modul Datenaufbereitungssystem benötigt. Dessen Aufgabe besteht in der Zusammenstellung und Transformation aller Daten, welche bei der Replikation zum Fat Client beziehungsweise bei dessen Synchronisation eine Rolle spielen. Dafür hat es direkten Zugriff auf alle der Model-Ebene zugehörigen internen Persistierungselemente.

Datenverteilungssystem: Wie das zuvor erläuterte Modul ist auch das Datenverteilungssystem nur im dezentralen Fall von Bedeutung und führt dann die eigentliche Replikation sowie Synchronisation der relevanten Daten zwischen dem Server und den Fat Clients aus. Mögliche konkrete Verfahren sowie Strategien zur Vermeidung von Konflikten sind in [Lie03] beschrieben, ein intern gekoppeltes Persistierungselement übernimmt die Speicherung der notwendigen Verbindungsparameter zu den Fat Clients.

Integrator: Das Modul Integrator dient sowohl der Einbindung von externen Daten für die Nutzung im Produktkonfigurator als auch der Bereitstellung von Informationen, welche aus dem Konfigurationsprozess entstehen und nach der Klassifizierung in Abschnitt 2.3.4.6 als globale Integrationsdaten bezeichnet werden. Damit prägt jenes Modul den integrativen Charakter des Grundmodells. Für die Verwaltung von Metadaten bezüglich der jeweiligen Schnittstellen kommen wie beim Datenverteilungssystem auch hier die intern gekoppelten Persistierungselemente zum Einsatz.

2.4.2 Abhängigkeiten

Aufgrund des modularen Charakters im Grundmodell existieren zwischen den in Abschnitt 2.4.1 beschriebenen Modulen einige Abhängigkeiten. Diese jeweils von einem konkreten Modul ausgehenden Zugriffe korrespondieren in Abbildung 2.11 mit den als Zugriffsrichtung gekennzeichneten Verbindungen und werden hier erläutert.

2.4.2.1 Steuerung

Bedingt durch die vielfältige Aufgabenstruktur des Steuerungsmoduls besitzt es als zentrale Einheit zu jedem anderen Modul im Grundmodell eine Abhängigkeit. Neben der Überwachung und Steuerung von komplexen Arbeitsabläufen über verschiedene Module hinweg sind diese vielseitigen Zugriffsmöglichkeiten auch notwendig, um die potentielle Erweiterbarkeit des Grundmodells zu gewährleisten.

2.4.2.2 Interaktionswerk

Sowohl die Datenpflege als auch die Konfiguration als die beiden wichtigsten in Abschnitt 2.3.3 dargestellten Prozesse eines Konfigurationssystems basieren auf der Kommunikation mit dem Nutzer über das Interaktionswerk. Dementsprechend erfolgt die Weitergabe der darin registrierten Eingaben entweder an das **Pflegesystem**, beispielsweise bei Hinterlegung eines Beschreibungstextes und Preises für eine Komponente, oder an das **Dialogwerk**, wenn beispielsweise eine Selektion im Rahmen einer Zusammenstellung stattfindet.

2.4.2.3 Pflegesystem

Die im Modul Pflegesystem stattfindende Datenmodellierung unterliegt insgesamt drei Abhängigkeiten. Einerseits muss eine Übermittlung von Informationen an das **Interaktionswerk** stattfinden, wenn beispielsweise der Nutzer für eine gepflegte Komponente alle Eigenschaften angezeigt bekommen möchte. Im Fall einer autorisierten Nutzung des Pflegesystems ergibt sich eine zusätzliche Abhängigkeit zum **Berechtigungssystem**, dessen Abfrageschnittstelle für die Ermittlung der Zugriffsrechte eines Nutzers dient. Schließlich existiert eine als beidseitige Implementierungsabhängigkeit bezeichnete Verbindung zum **Inferenzwerk**, da sich die Realisierung von Prüftechniken einerseits (siehe Abschnitt 2.3.5.4) sowie die zur Verfügung stehenden Konstrukte zur Formalisierung und Beschreibung von Abhängigkeiten andererseits gegenseitig beeinflussen.

2.4.2.4 Dialogwerk

Gemäß der funktionalen Beschreibung des Moduls Dialogwerk muss dieses einen Zugriff auf das **Interaktionswerk** besitzen, um die Versorgung mit allen während des Konfigurationsprozesses generierten Hinweisen und Meldungstexten gewährleisten zu können, welche dann im Interaktionswerk dargestellt werden. Aufgrund der Tatsache, dass jene Texte beispielsweise in einem Konfliktdialog durch das Ergebnis der Abhängigkeitsprüfung beeinflusst werden, existiert zudem eine Verbindung zum **Inferenzwerk**, um diesem fortlaufend die vom Nutzer vorgenommenen Eingaben und Selektionen in passender Art und Weise zu übermitteln.

2.4.2.5 Inferenzwerk

Auf Basis der entgegengenommenen Informationen zu ausgewählten Komponenten kann das Inferenzwerk die Gültigkeit des aktuellen Konfigurationszustands prüfen und daraus resultierende Einschränkungen oder Elemente zur Erzeugung von Fehlermeldungen an das Modul **Dialogwerk** übermitteln. Zusätzlich existiert die bereits erläuterte beidseitige Implementierungsabhängigkeit zum **Pflegesystem** bezüglich der gleichartigen Techniken zur Definition und Prüfung von Abhängigkeitsbeschreibungen.

2.4.2.6 Protokollierungssystem

Die ereignisbasierte Erfassung von Daten und Nutzeraktivitäten hinsichtlich der funktionalen Beschreibung des Protokollierungssystems erfordert den Zugriff auf das Modul **Dialogwerk**. Dadurch stehen potentiell alle zur späteren Auswertung relevanten Informationen zur Verfügung, welche im Laufe der Verarbeitung noch geeignet transformiert werden können.

2.4.2.7 Datenaufbereitungs- und Datenverteilungssystem

Beide Module stehen wechselweise in einer direkten Zugriffsabhängigkeit. Auf der einen Seite muss das Datenaufbereitungssystem nach Zusammenstellung der zu replizierenden Daten diese an das Datenverteilungssystem weiterleiten, um damit letztlich die verbundenen Fat Clients zu versorgen. Von diesen erhält das Datenverteilungssystem im umgekehrten Fall der Synchronisation die lokal erfassten Informationen, welche dann mit Hilfe des Datenaufbereitungssystems in den zentralen Persistierungselementen gespeichert werden.

2.4.3 Datenfluss und Persistierung

Ergänzend zu den in Abschnitt 2.4.2 diskutierten Abhängigkeiten zwischen einzelnen Modulen werden hier deren Beziehungen bezüglich der Persistierungselemente näher betrachtet. Dabei lassen sich die Varianten des (nur) lesenden und des schreibenden Zugriffs entsprechend der Aufgaben eines Moduls im Rahmen der vorgestellten Prozesse (siehe Abschnitt 2.3.3) unterscheiden. Diese Prozesse bilden mit den weiteren Klassifikationskategorien der Architektur und Persistierungsform aus Abschnitt 2.3.2 die in **Tabelle 2.2** gezeigte Matrix von verschiedenen Kombinationsmöglichkeiten.

	zentral		dezentral	
Prozess	Datenbank	Dateisystem	Datenbank	Dateisystem
Modellierung / Pflege	++	−	−−	−−
Bereitstellung	+	++	−	++
Konfiguration	++	−−	−	++

++: gut geeignet +: geeignet −: bedingt geeignet −−: ungeeignet

Tabelle 2.2: Verträglichkeit von Prozessen, Persistierungsarten und Architekturszenarien

Die Bewertung der einzelnen Varianten basiert auf den bisherigen Analysen sowie den Ergebnissen in [Lie11c]. Dabei lässt sich feststellen, dass die Nutzbarkeit einer Datenbank unabhängig vom konkreten Prozess direkt mit dem zentralen Architekturszenario korreliert. Weiterhin sollte aufgrund des potentiellen Mehrbenutzerbetriebs auch für den Prozess der Modellierung nur eine Datenbank eingesetzt werden. Dagegen ist für die Bereitstellung in jedem Fall das Dateisystem ausreichend, da der Konfigurationsprozess selbst nur lesend auf die Wissensbasis zugreifen muss. Schließlich eignet sich für die Speicherung von Konfigurationsergebnissen im zentralen Architekturszenario eine Datenbank sehr gut, während für dezentrale Clients auch die Ablage dieser Informationen im Dateisystem bereits die meisten Anforderungen erfüllt.

2.4.3.1 Lesende Zugriffe

Aufgrund der Modulbeschreibung in Abschnitt 2.4.1 ergibt sich nur für das Interaktionswerk, das Inferenzwerk und das Dialogwerk eine ausschließlich lesende Verbindung zum gemeinsamen Persistierungselement der Modell- und Oberflächendaten. Während das Interaktionswerk nur Daten zum Aufbau der Benutzeroberfläche benötigt, liest das Inferenzwerk lediglich die Modelldaten zur Prüfung von Abhängigkeiten. Schließlich verwendet noch das Dialogwerk beide Informationsarten, um passend zum Konfigurationszustand Textelemente in die Benutzeroberfläche eingebettet erzeugen zu können.

2.4.3.2 Schreibende Zugriffe

Den Großteil aller Module im Grundmodell kennzeichnet ein kombinierter Lese- und Schreibzugriff. Während das Pflegesystem, das Berechtigungssystem und das Protokollierungssystem jeweils nur eines der Persistierungselemente Modell- und Oberflächendaten, Administrationsdaten sowie Konfigurations- und Verkehrsdaten zur Verwaltung von Informationen nutzt, hat das Datenaufbereitungssystem zur Gewährleistung der vollständigen Replikation und Synchronisation Zugriff auf alle drei genannten internen Datenspeicher. Eine besondere Rolle spielen die Integrator-Module, da sie für jedes Persistierungselement eine bidirektionale Verbindung zur entsprechenden externen Datenquelle verwalten können.

2.5 Herausforderungen der Globalisierung

Im bisherigen Verlauf des Kapitels stand ausgehend von der Individualisierung die Einführung und Beschreibung von Produktkonfiguratoren als adäquates Mittel zur Beherrschung von Komplexität im Vordergrund. Wie einleitend in Abschnitt 1.1 motiviert, bestimmt daneben auch der Globalisierungs-Prozess unsere heutige Zeit, wobei im Rahmen der vorliegenden Arbeit insbesondere der Einfluss auf informationsverarbeitende Systeme von Interesse ist. Deswegen werden hier nach einer kurzen Begriffsklärung durch Abschnitt 2.5.1 die allgemeinen software-technischen Konsequenzen der Globalisierung in Abschnitt 2.5.2 betrachtet. Zusätzlich ergeben sich für Konfigurationssysteme bezüglich der zu konfigurierenden Produkte inhaltliche Auswirkungen, welche anschließend in Abschnitt 2.5.3 im Vordergrund stehen. Ausgehend von diesen Herausforderungen erfolgt in Abschnitt 2.5.4 die Präsentation eines Ansatzes zur Globalisierung von Modelldaten, bevor die gewonnenen Erkenntnisse in Abschnitt 2.5.5 kurz zusammengefasst werden.

2.5.1 Begriffsprägung

Wie in [Göb09] beschrieben, lassen sich prinzipiell drei Möglichkeiten zur marktspezifischen Anpassung eines Software-Produkts differenzieren. Dazu wird beim *individuellen* Ansatz für jeden Markt eine eigenständige Ausprägung der Anwendung implementiert, die vollkommen isoliert von den parallel existierenden Alternativen gepflegt werden muss. Dagegen basiert die *monolithische* Variante auf einem gemeinsamen Programmcode, welcher die Anforderungen der Märkte durch Codeabschnitte der Form „Wenn Ausführung im Markt A, dann springe zu Funktion X“ realisiert. Dadurch kann der Wartungsaufwand für zentrale Programmteile erheblich reduziert werden. Bei konsequenter Fortsetzung dieses Prinzips erhält man schließlich den *modularisierten* Ansatz, welcher bereits im Designprozess der Software marktspezifische und marktunabhängige Bereiche differenziert. Nachfolgend werden gemäß [HBL02] die gängigen Begriffe bei der Anwendung des modularisierten Ansatzes erläutert, deren Zusammenhänge in **Abbildung 2.12** dargestellt sind.

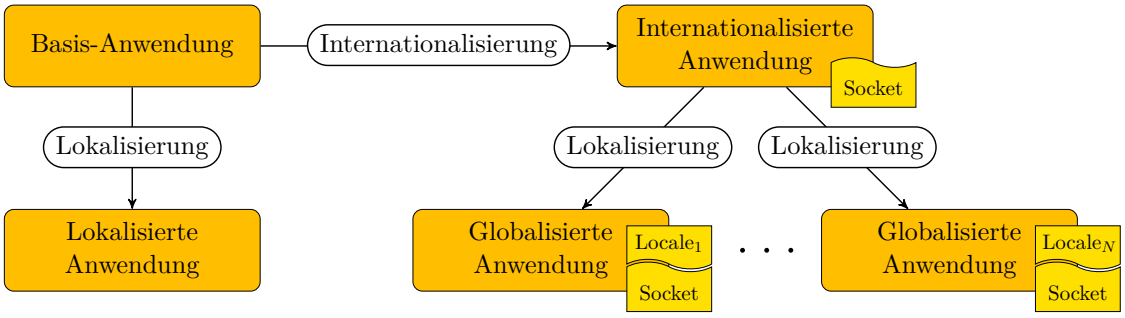


Abbildung 2.12: Internationalisierung und Lokalisierung von Software nach [Wes92]

2.5.1.1 Locale

Unter dem Begriff der *Locale* werden typischerweise Festlegungen für die Mensch-Maschine-Kommunikation bezüglich einer politischen, geografischen oder kulturellen Region zusammengefasst. Insbesondere sind davon die in Abschnitt 2.5.2 beschriebenen Merkmale wie der Zeichensatz oder die Schriftrichtung von Sprachen sowie Währungs- und

Zahlenformate betroffen. Die Benennung von Locales ist in [ISO02, ISO07] geregelt und orientiert sich an einem zweistelligen Kürzel für die vorherrschende Sprache und optional für das Land im Sinne eines Marktes. Beispielsweise repräsentiert `fr_CH` die französischsprachige Schweiz und `en_US` das amerikanische Englisch.

2.5.1.2 Internationalisierung

Die Strukturierung einer Software in allgemeingültige und Locale-spezifische Teile im Rahmen der Entwicklung oder Anpassung des Programmcodes wird als *Internationalisierung* definiert. Die aus diesem Prozess resultierende internationalisierte Anwendung kann ohne weitere Überarbeitung in Verbindung mit unterschiedlichen Locales eingesetzt werden. Dabei dienen sogenannte *Sockets* der Anbindung Locale-spezifischer Daten.

2.5.1.3 Lokalisierung

Die *Lokalisierung* einer nicht notwendigerweise internationalisierten Anwendung beschreibt deren Adaption an eine beliebige Locale. Üblicherweise versteht man darunter die Übersetzung der Oberflächentexte und die Änderung von Bildern oder Farben sowie Einstellungen bezüglich numerischer Werte und Währungen. Letztlich wird also eine gemäß den kulturellen und sprachlichen Gegebenheiten lokalisierte Anwendung erzeugt.

2.5.1.4 Globalisierung

Die *Globalisierung* fasst die beiden zuvor genannten Prozesse zusammen, d.h. eine globalisierte Anwendung wurde zunächst internationalisiert und anschließend lokalisiert.

2.5.2 Technische Konsequenzen

Nachdem im Abschnitt 2.5.1 die notwendigen Schritte zur Globalisierung eines Anwendungsprogramms aufgezeigt wurden, sollen hier basierend auf [Göb09] die resultierenden Konsequenzen aus technischer Sicht kurz betrachtet werden. Dies betrifft primär Fragen der Darstellung, Verarbeitung und Speicherung von Text aufgrund der herausragenden Bedeutung von Sprache für eine Locale. Im Fall der Produktkonfiguratoren sind dadurch insbesondere die Klassifikationskategorien der Technologie (siehe Abschnitt 2.3.2) und der Daten (siehe Abschnitt 2.3.4) betroffen.

2.5.2.1 Textdarstellung

An der Benutzeroberfläche darzustellender Text ist verknüpft mit zwei wesentlichen sprachabhängigen Eigenschaften, welche mehr oder weniger großen Einfluss auf das gesamte Layout inklusive der Ausrichtung von Elementen wie Beschriftungen, Funktionsknöpfen oder Menüs haben. Einerseits variiert die *Textlänge* für semantisch gleiche Wörter oder Wortgruppen zwischen einzelnen Sprachen, andererseits ist mit dieser auch die *Textrichtung* verbunden. Hier seien beispielhaft die gespiegelte Leserichtung für arabische Schriften und die mögliche, in der IT jedoch unübliche, vertikale Anordnung für Japanisch genannt.

2.5.2.2 Textverarbeitung

Auch die Interaktion und systeminterne Verarbeitung von Text unterliegt der Charakteristik einer Sprache. Dies betrifft bereits die *Texteingabe*, welche auf die Systemperipherie angewiesen ist und zumindest bei asiatischen Sprachen mit Tausenden von Zeichen nur mittels der geeigneten Zusammenfassung von Tastaturanschlägen gelingt. Eine in diesem Kontext häufig arbeitende *Rechtschreibprüfung und Silbentrennung* muss ebenfalls während der Lokalisierung sprachspezifisch bereitgestellt werden. Schließlich sind auch die Regeln für die *Sortierfolge* abhängig von dem jeweils verwendeten Zeichensatz.

2.5.2.3 Textspeicherung

Grundlage für die korrekte Speicherung von Text, aber auch für die beiden zuvor erläuterten Prozesse ist die Festlegung eines *Zeichensatzes*. Dieser bildet eine Anzahl von Zeichen (Glyphen) eindeutig auf die Menge der positiven Zahlen (Codepoints) ab und ermöglicht so eine Verarbeitung auf Basis von numerischen Werten. Ein bekannter Vertreter derartiger Zeichenkodierungen ist der Unicode-Standard (z.B. UTF8), welcher eine universelle Kodierung jedes potentiellen Schriftzeichens in weltweit allen Sprachen unterstützt [All06].

2.5.2.4 Datenformatierung

Parallel zur Darstellung, Verarbeitung und Speicherung sprachabhängiger Texte sind mit einer Locale auch verschiedene landesspezifische Konventionen zur Formatierung von Daten verbunden. Dies betrifft einerseits die Anzeige von Datumsangaben und Zeitangaben, selbst innerhalb eines Kalenderformats wie des weltweit angewandten gregorianischen Kalenders. Andererseits sind sowohl die Struktur von Adressen oder Telefonnummern als auch das Einheitensystem und die zur Zahlendarstellung verwendeten Trennzeichen von der Locale abhängig. Soll der Nutzer bei der Eingabe derartiger Informationen durch eine Syntaxprüfung unterstützt werden, muss diese ebenfalls lokalisiert werden. Einige Beispiele zu den genannten Abweichungen sind in **Tabelle 2.3** für die Locales `en_US` und `de_DE` dargestellt.

Datenformatierung	USA (`en_US`)	Deutschland (`de_DE`)
Datumsangabe	04/11/2008	11.04.2008
Zeitangabe	1:23 PM	13:23
Adressdaten	414 East Clark Street Vermillion, SD 57069	Humboldtstraße 13 07743 Jena
Telefonnummer	1-605-677-5321	+49 (0)3641/28440
Währung	$ 1,000.00	1.000,00 EUR
Maßeinheiten	imperial (Fuß/Pfund)	metrisch (Meter/Gramm)

Tabelle 2.3: Vergleich von Datenformatierungen nach [Göb09]

2.5.3 Inhaltliche Konsequenzen

Während die Lokalisierung von Standard-Anwendungen wie Internet-Browser oder Textverarbeitungsprogramme nur die in Abschnitt 2.5.2 beschriebenen Merkmale betrifft, hat jener Prozess bei Produktkonfiguratoren auch Einfluss auf das Produkt-Portfolio, welches durch die Modelldaten (siehe Abschnitt 2.3.4.1) repräsentiert wird. Diese müssen sowohl bezüglich der Struktur als auch der Eigenschaften von Komponenten marktabhängig angepasst werden können. Hierbei ist ein Markt bzw. Region als Teil der in Abschnitt 2.5.1.1 definierten Locale zu verstehen.

2.5.3.1 Struktur der Modelldaten

Bedingt durch gesetzliche Vorschriften, abweichende Marketing-Strategien oder die Berücksichtigung verschiedener Einkommensstrukturen beziehungsweise Lebensstile in den jeweiligen Märkten kann einerseits die Verfügbarkeit einzelner Produktkomponenten und andererseits deren Zusammensetzung variieren. Damit verbunden sind meist auch Änderungen in den Abhängigkeitsdefinitionen und den Selektionskriterien. Allerdings zeigen Erfahrungen aus der Praxis, dass innerhalb eines mehrsprachigen Marktes wie der Schweiz[6] durch eine Sprache zwar die textuellen Beschreibungen von Produktkomponenten, aber nicht deren Struktur oder Existenz beeinflusst werden [Göb09].

2.5.3.2 Eigenschaften der Modelldaten

Analog zur Lokalisierung der Benutzeroberfläche eines Produktkonfigurators sollten auch dessen Modelldaten-Eigenschaften für den Einsatz in heterogenen Märkten angepasst werden. Hiervon betroffen sind alle anzeigerelevanten Merkmale von Produktkomponenten und Selektionskriterien, wie beispielsweise Beschreibungstexte, verknüpfte Dokumente und Preise, wohingegen interne Flags und Attribute unverändert bleiben. Die weiteren Betrachtungen beschränken sich auf die sprachlichen Aspekte als prägendstes Elemente einer Locale, allerdings sind andere Auswirkungen beispielsweise auf die Preisbildung in ähnlicher Weise vorstellbar.

Abhängigkeitstyp	marktunabhängig	marktspezifisch
sprachunabhängig	**neutral** z.B. Komponenten-ID	**nur marktspezifisch** z.B. Währung/Preise
sprachspezifisch	**nur sprachspezifisch** z.B. Beschreibungstext	**Locale-spezifisch** z.B. Sprachvarietät/Dialekt

Tabelle 2.4: Abhängigkeitsmatrix für Modelldaten-Eigenschaften nach [Göb09]

Wie in **Tabelle 2.4** dargestellt, können die Eigenschaften von Modelldaten bezüglich ihrer Abhängigkeit von Sprache und Markt als den beiden Teilen einer Locale in vier Gruppen klassifiziert werden. Zu den vollständig unabhängigen und damit neutralen Eigenschaften zählen alle für die interne Verarbeitung notwendigen Attribute, beispielsweise die ID einer Produktkomponente. Dagegen sind deren beschreibende Texte in aller Regel nur

[6] Offizielle Amtssprachen in der Schweiz sind Deutsch, Französisch, Italienisch und Rätroromanisch.

sprachabhängig, d.h. in allen Märkten mit dieser Sprache gleich. Ausnahmen hierzu bilden Locale-spezifische Sprachvarietäten oder Dialekte, so beispielsweise die synonymen Begriffe „Abitur“ in Deutschland (`de_DE`) und „Matura“ in Österreich (`de_AT`). Zuletzt lassen sich noch marktspezifische Merkmale wie Währungen identifizieren, die in einem Markt unabhängig von dessen offiziellen Sprachen gelten.

2.5.4 Realisierungsansatz

Ausgehend von den in Abschnitt 2.5.3 beschriebenen Konsequenzen soll mit diesem Abschnitt ein Ansatz für die Globalisierung von Modelldaten in Produktkonfiguratoren präsentiert werden, deren Persistierung entsprechend der Tabelle 2.2 in einem Datenbanksystem erfolgt. Dabei verwendete Konzepte sind Bestandteil eines in [Göb09] spezifizierten Prototyps, der eine Weiterentwicklung des Produktkonfigurators CREALIS der Firma ORISA Software GmbH[7] bezüglich Internationalisierung beinhaltet. Beispielhaft stehen hier die Modellierung von Eigenschaften sowie die Unterstützung von Standardsprachen im Mittelpunkt der Betrachtungen. Darüber hinaus ist der vollständige Entwurf zur Internationalisierung aller in Abschnitt 2.3.4.1 klassifizierten Modelldaten sowie deren Prozessierung und Visualisierung durch ein Konfigurationssystem in [Göb09] zu finden.

2.5.4.1 Internationalisierung von Eigenschaften

Wie in Abschnitt 2.5.3.2 dargestellt, besitzen Modelldaten abhängig vom Objekttyp eine Menge von Eigenschaften, deren Werte jeweils durch folgende Attribute eindeutig bestimmt sind:

Objekt: Hiermit ist sowohl die eindeutige Zuordnung zu einem konkreten Objekt im Sinne eines Datensatzes gemeint als auch eine dadurch implizite Klassifizierung innerhalb der verschiedenen Objekttypen wie beispielsweise Komponenten, Merkmale oder Abhängigkeitsbeschreibungen.

Eigenschaft: Die Zuweisung eines Eigenschaftswertes erfolgt bezüglich einer konkreten Eigenschaft, welche für einen Objekttyp eindeutig benannt sei muss (z.B. Name oder Preis einer Komponente).

Sprache (optional): Liegt aus semantischer Sicht für die betreffende Eigenschaft eine Sprachspezifik vor, dann sind dessen Werte sprachabhängig zu speichern.

Markt (optional): Analog zur Sprache müssen Eigenschaftswerte im Fall einer Marktabhängigkeit unter Zuordnung eines konkreten Marktes gespeichert werden.

Wie später in Abschnitt 3.3 näher erläutert, sollen aus Gründen der Redundanzfreiheit und Unabhängigkeit die internationalisierten Daten nicht mit dem ursprünglichen fachlichen Datenmodell verwoben werden. Stattdessen lassen sich gemäß [Göb09] die beschriebenen Attribute eines Eigenschaftswertes auch als dessen Dimensionen betrachten. Zur Verwaltung derartiger multidimensionaler Strukturen haben sich seit vielen Jahren Data Warehouses zur Unterstützung des Online Analytical Processing (OLAP) bewährt [BG08].

[7] `http://www.orisa.de`

Dabei kommt am häufigsten das Konzept des *Star-Schemas* zum Einsatz, um jene multidimensionalen Informationen auf die flachen Strukturen eines (relationalen) Datenbanksystems abzubilden. Ein solches Star-Schema enthält beliebig viele Dimensionstabellen, welche über Schlüssel mit einer zentralen Faktentabelle verbunden sind und einen Fakt eindeutig beschreiben [KR02]. Angewendet auf den hier zu modellierenden Sachverhalt können die vier aufgezählten Attribute als Dimensionsdaten aufgefasst werden, während die Eigenschaftswerte den Faktendaten entsprechen.

Eigenschaft	Erläuterung	marktspezifisch	sprachspezifisch
ModuleID	internes Identifikationsmerkmal	–	–
Key	Artikelcode der Komponente	–	–
UserCreate	Name/ID des Erstellers	–	–
DateCreate	Zeitpunkt der Erstellung	–	–
Name	Bezeichnung der Komponente	–	✓
Description	Beschreibung zur Komponente	–	✓
Price	Preis der Komponente	✓	–

Tabelle 2.5: Eigenschaften von Komponenten

Zur Veranschaulichung der weiteren Ausführungen seien Komponenten (eng. *module*) stellvertretend für Modelldaten betrachtet, einige typische Eigenschaften sind in **Tabelle 2.5** beispielhaft dargestellt. Dabei sind ModuleID, Key, UserCreate und DateCreate den neutralen Eigenschaften im Sinne der Klassifizierung durch Tabelle 2.4 zuzuordnen, wohingegen Name, Description und Price sprach- bzw. marktabhängig sein sollen.

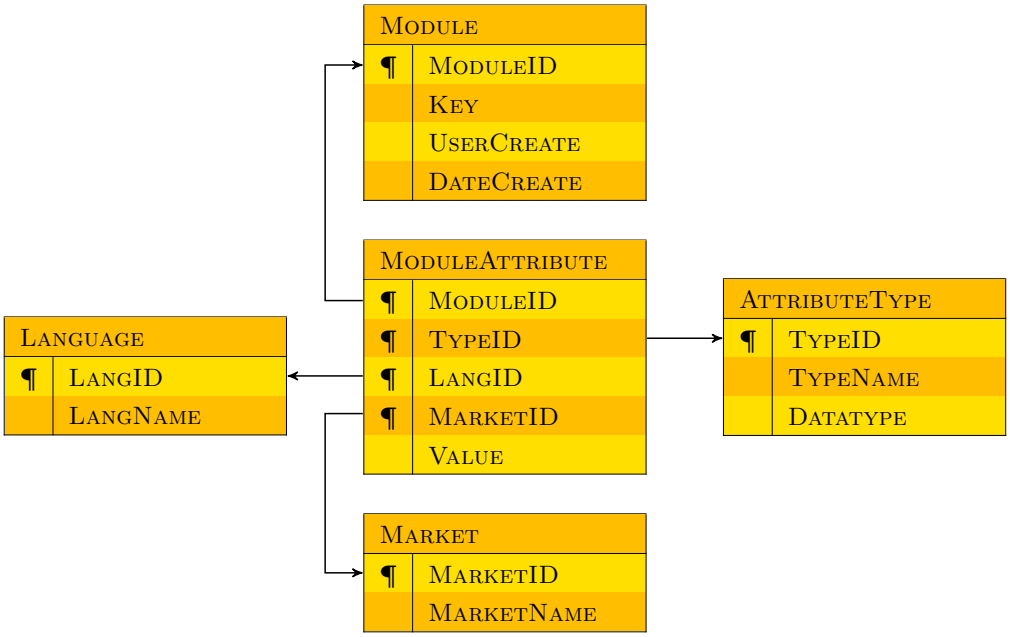


Abbildung 2.13: Modellierung von Komponenten-Eigenschaften nach [Göb09]

Der entscheidende Gedanke beim Übertragen der Konzepte des Star-Schemas auf die in **Abbildung 2.13** skizzierte Modellierung von Produktkomponenten besteht darin, alle

neutralen Eigenschaften von Komponenten unverändert in ihrer ursprünglichen Struktur (MODULE) zu belassen und nur die tatsächlich lokalisierten Eigenschaftswerte in der Faktentabelle (MODULEATTRIBUTE) abzubilden. Schließlich komplettieren die Sprache (LANGUAGE), der Markt (MARKET) und die Eigenschaftsart (ATTRIBUTETYPE) als Dimensionstabellen die zu Beginn des Abschnitts aufgezählten Attribute.

Allerdings sind mit der resultierenden Modellierung einige Herausforderungen verbunden, die nachfolgend kurz motiviert werden sollen. Eine ausführliche Betrachtung sowie die Diskussion alternativer Lösungsmöglichkeiten bilden in [Göb09] einen Schwerpunkt.

Eigenschaften-Trennung: Basierend auf der Klassifizierung in neutrale und dimensionsabhängige Eigenschaften wurde die logische Datenstruktur für Komponenten in zwei Teile getrennt (MODULE, MODULEATTRIBUTE), was sowohl die fachliche Datenmodellierung als auch den anwendungsseitigen Zugriff komplexer gestaltet.

Dimensions-Optionalität: Der Aufbau von MODULEATTRIBUTE erfordert die Angabe einer Sprache *und* eines Marktes bei Speicherung von Werten. Die Abbildung der in Tabelle 2.4 klassifizierten nur sprach- oder marktabhängigen Werte gelingt entweder durch Konzepte wie Surrogatschlüssel bzw. Schlüssel-Entfernung oder mit Hilfe von zusätzlichen Faktentabellen geringerer Dimensionalität.

Modelldaten-Objektart: Abbildung 2.13 stellt nur ein Konzept für Eigenschaften von Komponenten dar. Andere Modelldaten (siehe Abschnitt 2.3.4.1) können durch eine zusätzliche Objekttyp-Dimension oder weitere Faktentabellen integriert werden.

Eigenschaftswert-Datentyp: Abhängig vom lokalisierten Eigenschaftstyp müssen in MODULEATTRIBUTE.VALUE Texte, Zahlen oder andere Datentypen gespeichert werden, was mittels typspezifischer Attribute oder Faktentabellen realisierbar ist.

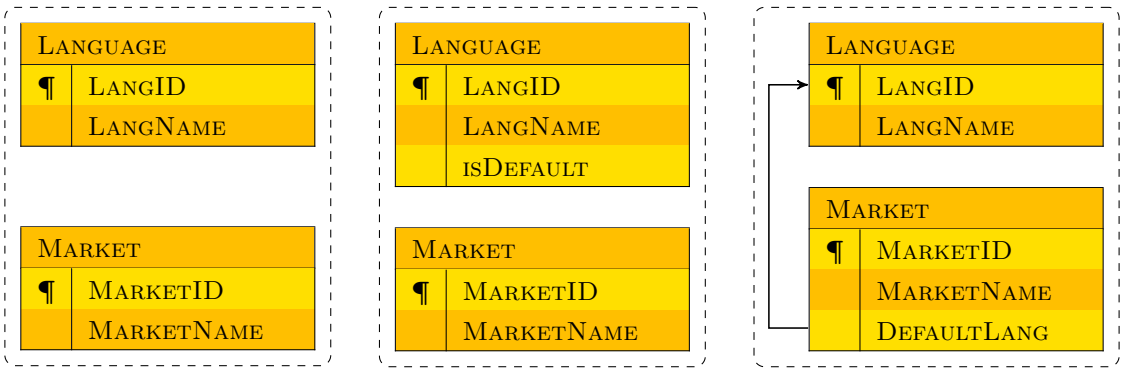
2.5.4.2 Modellierung von Standardsprachen

Aufgrund der Trennung von neutralen und dimensionsabhängigen Eigenschaften entsprechend der Modellierung in Abbildung 2.13 wird ein Konzept benötigt, um im Fall nicht vorhandener bzw. nicht gepflegter Eigenschaftswerte reagieren zu können. Hierfür lässt sich eine sogenannte *Standardsprache* definieren, zu der alle sprachspezifischen Eigenschaften im Produktkonfigurator gepflegt sein müssen und auf die dann bei fehlenden Informationen zu einer anderen Sprachen zurückgegriffen werden kann. Dabei ist laut [Göb09] ein globaler von einem lokalen Wirkungsbereich zu unterscheiden.

Globale Standardsprache: Innerhalb des Produktkonfigurators gibt es nur genau eine Standardsprache, welche für sämtliche unterstützten Locales und deren verbundenen Märkte gilt. Typischerweise kommt dafür Englisch (z.B. `en_GB`) aufgrund der enormen weltweiten Verbreitung und Akzeptanz als offizielle Geschäftssprache internationaler Organisationen zum Einsatz.

Lokale Standardsprache: Im Gegensatz zur globalen Ausprägung wirkt die lokale Standardsprache nur innerhalb *eines* Marktes. Sinnvollerweise sollte dies eine dem Markt zugeordnete Sprache sein, womit für diesen Ansatz nur mehrsprachige Märkte in Frage kommen. Zudem lassen sich die Konzepte der globalen und lokalen Standardsprache miteinander verbinden, wobei dann letztere prinzipiell Vorrang hat.

Neben der anwendungsseitigen Implementierung von Funktionalität zur Auswertung einer Standardsprache kann zumindest die Definition des soeben klassifizierten Wirkungsbereichs durch eine entsprechende Modellierungserweiterung realisiert werden. Die entsprechenden Konsequenzen für die relevanten Dimensionstabellen zu Sprache und Markt sind in **Abbildung 2.14** dargestellt.



(a) keine (b) globale (c) lokale

Abbildung 2.14: Modellierung von Standardsprachen nach [Göb09]

Während Abbildung 2.14(a) den Ausgangszustand gemäß Abbildung 2.13 nochmals verdeutlicht, zeigt Abbildung 2.14(b) die Realisierung der globalen Standardsprache mittels Kennzeichnung durch Language.isDefault. Für die Zuordnung einer lokalen Standardsprache muss dagegen die Definition eines Marktes wie in Abbildung 2.14(c) skizziert um Market.DefaultLang erweitert werden.

2.5.5 Fazit

Wie der letzte Teil dieses Kapitels ansatzweise gezeigt hat, unterliegen Konfigurationssysteme im besonderen Maße dem Prozess der Globalisierung durch jene in Abschnitt 2.5.3 thematisierten Konsequenzen hinsichtlich marktspezifischer Modelldaten. Deren Internationalisierung wirkt sich direkt auf die zugrunde liegende Persistierung aus und birgt bezüglich der Abbildungstechniken die in [Göb09] detailliert betrachtete Komplexität und Vielfalt. Dabei ist zu beobachten, dass hierdurch das fachliche Datenmodell eines Produktkonfigurators zur Unterstützung der in Abschnitt 2.2 erläuterten zentralen Anforderungen stark beeinflusst wird.

Zusätzlich lassen sich neben der Internationalisierung noch weitere ähnlich gelagerte Systemeigenschaften identifizieren, welche nicht zum eigentlichen Funktionsumfang eines Produktkonfigurators gehören, wie beispielsweise der Zugriff auf versionierte Datenbestände. Ihnen allen ist gemein, dass sie eine querschnittliche Wirkung auf die gesamte Datenhaltung haben und dadurch die ursprüngliche Datenmodellierung zunehmend unübersichtlicher, unflexibler sowie komplexer wird. Um diesem Problem zu begegnen, ist ein entsprechendes Paradigma zur generischen und erweiterbaren Abbildung dieser querschnittlichen Belange notwendig, welches im nachfolgenden Kapitel 3 vorgestellt wird.

Kapitel 3

Aspektorientierte Datenhaltung

Dieses Kapitel stellt die Aspektorientierte Datenhaltung (AOD) als Paradigma vor, welches die generische Abbildung von querschnittlichen Belangen im Kontext einer datenbankbasierten Persistierung ermöglichen soll. Hierzu werden in **Abschnitt 3.1** die charakteristischen Merkmale jener sogenannten funktionalen Aspekte als zentraler Bestandteil vorgestellt und anschließend mit einem praxisnahen Anwendungsbeispiel in **Abschnitt 3.2** veranschaulicht. Zur Bewältigung der dabei identifizierbaren Herausforderungen im Rahmen der Datenhaltung werden in **Abschnitt 3.3** einige Anforderungen formuliert, welche gemeinsam das AOD-Paradigma kennzeichnen. Sowohl der **Abschnitt 3.4** zur Analyse verschiedener Technologien und Persistierungsmodelle für die AOD als auch deren Bewertung und Vergleich in **Abschnitt 3.5** runden das Kapitel ab.

3.1 Funktionale Aspekte

Die nachfolgenden Ausführungen dienen der grundsätzlichen Motivation und Problemeingrenzung hinsichtlich funktionaler Aspekte im Datenhaltungs-Kontext. Zu diesem Zweck wird in Abschnitt 3.1.1 eine begriffliche Herleitung und Definition vorgenommen, auf deren Basis anschließend in Abschnitt 3.1.2 die Klassifikation datenspezifischer Auswirkungen durch funktionale Aspekte stattfindet.

3.1.1 Begriffsprägung

Ein maßgebliches Konzept der Informationstechnologie ist die Modularisierung und Kapselung sowohl von Daten als auch Funktionalität in logische Einheiten, um die Implementierung und Wartung komplexer Systeme gewährleisten zu können [Lie10a]. Neben der Definition von Architekturmustern wie dem ANSI/SPARC-Modell [HR01] sind diese Bestrebungen auch in der historischen Entwicklung imperativer Programmiersprachen zu beobachten (von strukturiert über prozedural bis modular). Als das derzeit verbreitetste und praxisrelevanteste Paradigma basiert die Objektorientierte Programmierung (OOP) auf sogenannten Objekten, die eine inhaltliche Kapselung von Daten und darauf anwendbaren Funktionen ermöglichen. Eine wichtige Voraussetzung in diesem Zusammenhang ist die Festlegung einer zentralen Dimension, welche über die Einteilung der Objekte und Funktions-Zuordnung entscheidet [Pie11a]. Üblicherweise bilden die fachlichen Aspekte

des jeweiligen Anwendungsbereichs jenes Kriterium, wodurch im Prozess der Softwareentwicklung die Kapselung von Klassen und Komponenten entsprechend der Geschäftslogik erfolgt. Allerdings existieren in aller Regel weitere nicht-fachliche Anforderungen, die für eine Vielzahl der aus fachlicher Sicht modellierten Komponenten in gleichem Maße relevant sind und somit über diese verstreut abgebildet werden müssen. Typische Beispiele für jene sogenannten *querschnittlichen Belange* (engl. cross-cutting concern) stellen das Logging und die Fehlerbehandlung dar, welche prinzipiell in jeder Klasse und häufig nach einem wie in **Listing 3.1** skizzierten Schema implementiert werden müssen. Für die Bewältigung der daraus resultierenden Probleme, insbesondere einer schlechten Wiederverwendbarkeit und aufwändigen Wartung des Programmcodes, bietet die von [KLM+97] vorgestellte Aspektorientierte Programmierung (AOP) geeignete Techniken und Konzepte.

```
public long calculateFactorial (int a) {
    String methodName = "calculateFactorial";
    logger.doTrace ("Start of method " + methodName);

    long result = 1; // calculate the factorial value now
    for (int i=1; i<=a; i++) result = result * i;

    logger.doTrace ("End of method " + methodName);
    return result;
}
```

Listing 3.1: Beispiel für den Logging-Aspekt in Java-Programmcode

Das Phänomen der querschnittlichen Belange lässt sich in ähnlicher Art und Weise auch auf die Datenhaltung übertragen. Dort besteht die primäre Aufgabe, im Rahmen der Modellierung geeignete logische Datenstrukturen zur Abbildung von Informationen aus der Geschäftsrealität zu definieren und dadurch die Persistierung von sogenannten Business-Objekten sicherzustellen. Dabei können ebenfalls zusätzliche Aspekte identifiziert werden, die sich der gewählten fachlich geprägten Dimensionierung „widersetzen“ und entsprechend der folgenden Definition nach [Lie10a] charakterisierbar sind.

▶ **Definition 3.1** *Ein **funktionaler Aspekt** bezeichnet im Kontext dieser Arbeit eine spezifische Eigenschaft eines Datenmodells bzw. Gesamtsystems, welche sich nicht einer konkreten Datenstruktur zuordnen lässt, sondern eine Vielzahl der modellierten Objekte betrifft.*

Beispiele für funktionale Aspekte in der Datenhaltung sind unter anderem die Unterstützung von Mehrsprachigkeit wie in Abschnitt 2.5 erläutert sowie die Versionierung von Daten. Diesbezüglich hat ein funktionaler Aspekt natürlich nur dann eine querschnittliche Wirkung, wenn nicht bereits die fachliche Modellierung nach dessen Dimension ausgerichtet ist, wie dies beispielsweise für den Versionsaspekt im Kontext einer Versionsverwaltung der Fall wäre.

Analog zu querschnittlichen Belangen in der Softwareentwicklung führt deren Charakteristik auf der Persistierungsebene letztlich zu einem erhöhten Pflegeaufwand und einem geringen Wiederverwendungsgrad sowohl für fachliche als auch aspektspezifische Datenstrukturen aufgrund der fehlenden Modularisierung. Nachfolgend werden in Abschnitt 3.1.2 die konkreten Auswirkungen funktionaler Aspekte auf die Datenhaltung näher betrachtet.

3.1.2 Auswirkungen auf die Datenhaltung

Zur Persistierung von Daten, insbesondere in der datenbankbasierten Variante, sind neben den eigentlichen Inhalten („was“) auch die zugeordneten Strukturen („wie“) von Bedeutung. Dementsprechend können sich die Konsequenzen funktionaler Aspekte über beide Bereiche erstrecken. Anhand des relationalen Modells [Cod70] und einer exemplarischen Relation $R = (Att_1, Att_2, Att_3)$ lässt sich der Wirkungsbereich funktionaler Aspekte wie in **Abbildung 3.1** dargestellt veranschaulichen.

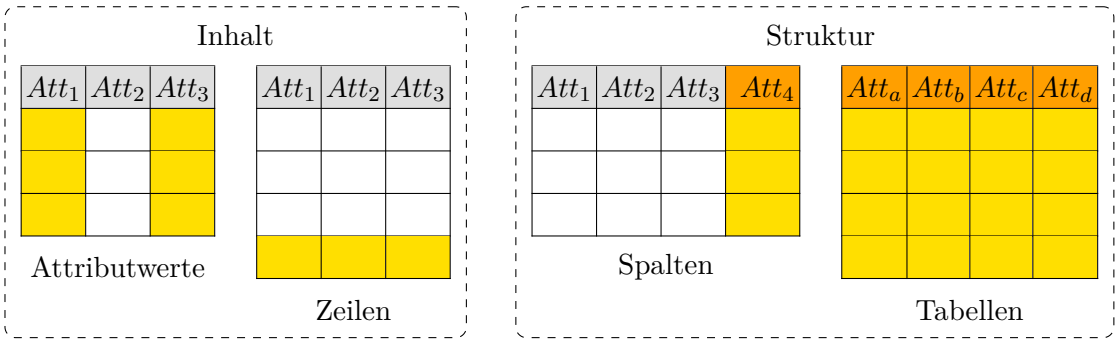


Abbildung 3.1: Auswirkungen funktionaler Aspekte auf die Datenhaltung

Dabei symbolisieren die farblichen Hervorhebungen im Fall der Attributwerte die mögliche Änderung *existierender* Werte unter Einfluss von Aspekten, wohingegen bei den gleichermaßen markierten Zeilen, Spalten und Tabellen bereits deren Existenz aspektabhängig sein kann. Eine genaue Analyse sowohl der inhaltlichen als auch der strukturellen Ebene erfolgt in den nächsten beiden Abschnitten.

3.1.2.1 Inhaltliche Ebene

Bereits im Zuge der in Abschnitt 2.5.3 diskutierten Globalisierung von Modelldaten eines Produktkonfigurators wurde deutlich, dass die Einflüsse von Markt und Sprache als Vertreter funktionaler Aspekte auf jede der Eigenschaften (Attribute) von Komponenten, Abhängigkeitsdefinitionen und anderen Objekten individuell wirken können und damit eine Klassifizierung gemäß Tabelle 2.4 erlauben. Die Generalisierung jener Erkenntnisse im Kontext des hier zugrunde gelegten relationalen Modells ist durch nachstehende Prämissen geprägt, welche gleichzeitig die bisherige Charakterisierung funktionaler Aspekte in Definition 3.1 hinsichtlich aspektspezifischer Attributwerte konkretisieren.

1. Funktionale Aspekte sind prinzipiell voneinander unabhängig und müssen sich deswegen in beliebigen Reihenfolgen in ein Datenmodell integrieren lassen.

2. Funktionale Aspekte besitzen eine endliche Zahl diskreter Ausprägungen, wie beispielsweise die Menge aller Locales für den Sprachaspekt laut [ISO02, ISO07].

3. Ein funktionaler Aspekt wirkt auf einzelnen Attributen Att_i beliebiger Relationen.

4. Ein Attribut Att_i kann von einer beliebigen Anzahl k funktionaler Aspekte beeinflusst werden, die Werte dieses Attributs sind dann von jenen k Aspekten abhängig.

Betrachtet man funktionale Aspekte als Abhängigkeits-Dimensionen, bietet sich für die Verwaltung der aspektspezifischen Attributwerte insbesondere unter Berücksichtigung der zuvor genannten vierten Voraussetzung ein sogenannter *k-dimensionaler Hyperquader*[1] als Datenmodell an [Sch11]. Dieser besitzt die notwendige Struktur zur Hinterlegung eines Attributwertes für jede potentiell mögliche Kombination von Ausprägungen jener Aspekte, welche das betreffende Attribut beeinflussen.

Die Visualisierung eines solchen Hyperquaders ist beispielhaft in **Abbildung 3.2** für die Relation $R = (Att_1, Att_2, Att_3, Att_4)$ skizziert. Aus Projektionsgründen wurde die Anzahl der gleichzeitig auf ein Attribut wirkenden Aspekte auf drei begrenzt, um sie als Dimensionen im euklidischen Raum darstellen zu können. Konkret sollen in der Relation R die Attribute Att_1 und Att_4 jeweils von zwei und Att_2 von drei Aspekten abhängig sein, während auf Att_3 nur ein Aspekt wirkt. Eine weiterführende Formalisierung funktionaler Aspekte findet in Kapitel 4 statt.

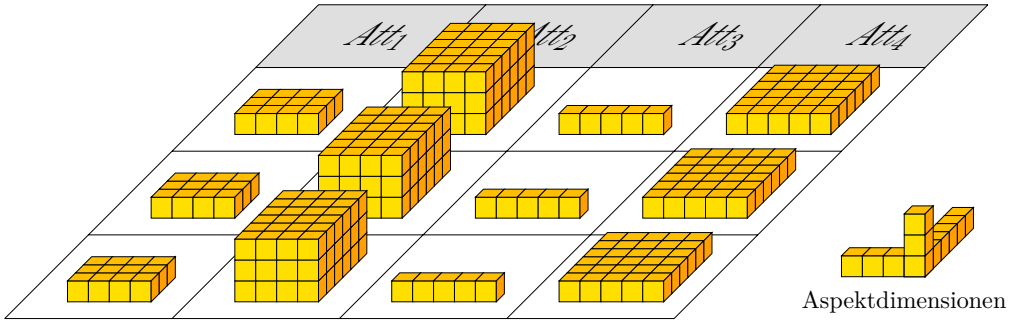


Abbildung 3.2: Hyperquader-Modell für aspektspezifische Attributwerte und drei Aspekte

Wie durch Abbildung 3.1 klassifiziert, können auf der inhaltlichen Ebene schließlich auch vollständige Objekte, modelliert als Zeilen von Relationen, in Abhängigkeit bestimmter Aspekte existieren oder durch diese ausgeblendet werden. Zur Bewältigung derartiger Auswirkungen funktionaler Aspekte muss es möglich sein, für jede beliebige Zeile einer Relation in Verbindung mit einer konkreten Kombination von Ausprägungen der in diesem Kontext gültigen Aspekte einen Wert zur Existenz jener Zeile zu hinterlegen. Ein praxisrelevanter Anwendungsfall für dieses Szenario sind die in Abschnitt 2.5.3.1 beschriebenen variablen und marktspezifischen Produktstrukturen. Begrenzt auf den Markt-Aspekt stellt beispielsweise das in [Göb09] vorgestellte Konzept der dreiwertigen Logik einen geeigneten Realisierungsansatz dar.

3.1.2.2 Strukturelle Ebene

Strukturelle Auswirkungen funktionaler Aspekte, welche sich hier sowohl durch unterschiedliche Attributmengen einer bestimmten Relation als auch anhand der Existenz von Relationen selbst äußern, erfordern eine große Flexibilität in der Datenhaltung. Diese auch in voller Konsequenz als Schemafreiheit generalisierte Anforderung ist unter anderem im Umfeld von Software as a Service (SaaS) ein zentrales Thema aufgrund der dort notwendigen Mandantenfähigkeit und wird deswegen im Rahmen dieser Arbeit nicht weiter betrachtet. Eine detaillierte Vorstellung und Bewertung einzelner Realisierungsansätze zu dieser Problematik bietet [AGJ+08].

[1] In Analogie zum Hyperwürfel ist damit ein zur Dimension k verallgemeinerter Quader gemeint.

3.2 Anwendungsbeispiel

Mit Hilfe eines einfachen an [Pie11b] angelehnten Beispiels sollen nachfolgend die Charakteristik und Probleme von funktionalen Aspekten bei der Datenhaltung veranschaulicht werden. Grundlage hierfür bilden die Produktkomponenten (Module) als kleiner Ausschnitt der in Abschnitt 2.3.4.1 erläuterten Modelldaten eines Produktkonfigurators. Deren exemplarische Modellierung anhand typischer Eigenschaften ist in Abschnitt 3.2.1 skizziert, anschließend findet in Abschnitt 3.2.2 eine Vorstellung der darauf wirkenden funktionalen Aspekte statt. Im Verlauf der Arbeit wird das beschriebene Anwendungsbeispiel an geeigneten Stellen zu Demonstrationszwecken aufgegriffen.

3.2.1 Modellierung

Abweichend zu den in Tabelle 2.5 (siehe Abschnitt 2.5.4.1) aufgeführten Eigenschaften seien Produktkomponenten im restlichen Verlauf der Arbeit durch das in **Abbildung 3.3** dargestellte Relationenschema MODULE repräsentiert.

MODULE	
¶	KEY [char]
	NAME [char]
	PRICE [decimal]
	DESCRIPTION [char]
	NORM* [char]
	MATERIAL* [char]
	USERCREATE [char]
	DATECREATE [date]
(¶)	ROWID [int auto]

Abbildung 3.3: Modellierung von Produktkomponenten

Eine Komponente lässt sich eindeutig über ihren alphanumerischen Schlüssel (KEY) identifizieren und besitzt einen Namen, einen Preis und eine Beschreibung. Weiterhin können optional eine Normierung sowie ein Material angegeben werden, für beide Attribute wurde zwecks Vereinfachung auf die Hinterlegung eigener Stammdatenrelationen und den Verweis per Fremdschlüssel verzichtet. Darüber hinaus sind sowohl der Nutzer als auch das Datum der initialen Erstellung vermerkt. Schließlich existiert parallel zum fachlichen Primärschlüssel KEY noch die interne ROWID als selbst hochzählendes ganzzahliges Schlüsselattribut, auf dessen Rolle noch genauer in Kapitel 4 eingegangen wird. Das Ausprägungs-Tupel in **Abbildung 3.4** wäre beispielsweise eine mögliche Komponente in dem soeben erläuterten Schema.

```
('MODSCR3141', 'Schraube', 0.45, 'Sechskant-Schraube 2mmx8mm',
  'BS 4190', 'Edelstahl', 'Matthias Liebisch', 2012-06-19, 3141)
```

Abbildung 3.4: Beispiel eines Komponenten-Tupels

3.2.2 Aspektdefinition

Als Konsequenz des in Abschnitt 2.5 diskutierten Prozesses der Globalisierung soll die in Abbildung 3.3 gezeigte beispielhafte Modellierung von Produktkomponenten unter dem Einfluss folgender funktionaler Aspekte stehen:

Sprache: Wie in Abschnitt 2.5.1.1 bereits erörtert, ist die Globalisierung gekennzeichnet durch die Anforderungen der verschiedenen Locales. Hierbei spielt insbesondere die Mehrsprachigkeit eine große Rolle, wodurch im Datenmodell die Persistierung von unterschiedlichen Sprachausprägungen für ein Objekt gewährleistet sein muss.

Region (Markt): Neben der Sprache können durch eine Locale auch Daten hinsichtlich einer Region im Sinne eines Marktes verändert werden. Sinnvolle Ausprägungen dieses Aspektes wären unter anderem nationale Märkte wie beispielsweise die Schweiz, Währungsräume im Sinne der Eurozone oder auch kontinentale Staatenverbünde wie die Europäische Union (EU).

Version: Aus Sicht der Produktentwicklung soll die Versionierung von Daten unterstützt werden. Hierbei setzt die Speicherung versionsabhängiger Daten eine endliche Menge globaler Versionsnummern voraus.

Staffel: Der Preis eines Produkts oder einer Komponente hängt in der Praxis meist von einer der Kundengruppe zugeordneten Rabattstaffel ab. Deshalb muss es möglich sein, dem Preisattribut eines Objekts zu jeder Ausprägung der Staffel einen individuellen Wert zuweisen zu können.

Der Wirkungsbereich jener vier Aspekte umfasst jedoch nicht alle Attribute der Tabelle Module. Vollständig unbeeinflusst sind der Schlüssel, die Erstellungs-Informationen sowie die interne RowID. Dagegen sind vom Aspekt der Sprache Name, Beschreibung und Material einer Komponenten abhängig, während sich die Region auf deren Norm und Preis auswirken kann. Letztere Eigenschaft ist zudem noch aspektspezifisch bezüglich der Staffel. Schließlich sollen alle mit Ausnahme der vier zuerst genannten Attribute versionierbar sein. Effektiv ist beispielsweise der Name einer Komponente von den Aspekten Sprache und Version abhängig, **Abbildung 3.5** bietet hierzu eine vollständige Übersicht.

	Sprache	Region	Version	Staffel
Key	–	–	–	–
Name	✓	–	✓	–
Price	–	✓	✓	✓
Description	✓	–	✓	–
Norm	–	✓	✓	–
Material	✓	–	✓	–
UserCreate	–	–	–	–
DateCreate	–	–	–	–
RowID	–	–	–	–

Abbildung 3.5: Relevanz funktionaler Aspekte für Attribute von Module

3.3 Anforderungen des AOD-Paradigmas

Die Bewältigung der in Abschnitt 3.1.1 beschriebenen negativen Auswirkungen funktionaler Aspekte auf ein Datenmodell erfordert letztendlich ein Lösungskonzept, welches hier als das Paradigma der Aspektorientierten Datenhaltung vorgestellt wird. Dessen Zielstellung ist die Unterstützung einer datenbankbasierten fachlichen Modellierung unabhängig von den zu integrierenden funktionalen Aspekten. Dabei sollen die nachfolgend aufgeführten Anforderungen und Eigenschaften gemäß [Lie10a] gelten.

Modularität: Trotz des in Definition 3.1 festgehaltenen querschnittlichen Charakters funktionaler Aspekte soll es möglich sein, sowohl die zu einem solchen Aspekt zugehörigen Metadaten und Information als auch alle von diesem beeinflusste Modelldaten innerhalb eigenständiger logischer Modellierungseinheiten zusammenzufassen. Dadurch ließe sich die problematische Vermischung mit fachlichen Strukturen vermeiden.

Orthogonalität: Die Integration von n verschiedenen Aspekten in ein Datenmodell soll zwei Bedingungen genügen, wobei ein konkretes Attribut von $[0, \ldots, n]$ Aspekten abhängig sein kann. Einerseits wird gefordert, dass alle $n!$ möglichen Reihenfolgen zur Integration der betrachteten Aspekte das betreffende Datenmodell jeweils in einen semantisch äquivalenten Zustand überführen. Daraus ergibt sich die freie Kombinierbarkeit von Datenmodellen und Aspekten. Andererseits darf es zu keinerlei Wechselwirkungen zwischen den beeinflussten Attributen kommen. Dies betrifft für zwei Aspekte A_1 und A_2 mit ihren Mengen abhängiger Attribute $\mathcal{M}(A_1)$ und $\mathcal{M}(A_2)$ sowohl die Attribute innerhalb einer solchen Menge $\mathcal{M}(A_i)$ als auch die von beiden Aspekten beeinflussten Attribute der Schnittmenge $\mathcal{M}(A_1) \cap \mathcal{M}(A_2)$.

Lokalität: Ergänzend zur Modularitäts-Eigenschaft, durch die gewisse Anforderungen an die Speicherung der aspektspezifischen Informationen und Metadaten gestellt werden, soll zudem die Administration funktionaler Aspekte eine möglichst hohe Lokalität aufweisen. Konkret ist damit die Minimierung struktureller Transformationen am Datenmodell (Schemaevolution) aufgrund der Integration eines funktionalen Aspekts oder dessen Abhängigkeitspflege verbunden.

Universalität: Das Paradigma der Aspektorientierten Datenhaltung soll ein generisches Lösungskonzept unabhängig sowohl von konkreten funktionalen Aspekten als auch dem Anwendungsbereich bieten. Zur Gewährleistung eines maximalen Einsatzgebiets in der Praxis sind für die Implementierung der AOD nur standardisierte Techniken zu verwenden, beispielsweise im Fall des relationalen Modells auf dem SQL:92-Sprachumfang [ISO92] basierende Modellierungskonstrukte.

Benutzerfreundlichkeit: Neben den zuvor aufgeführten eher technischen Anforderungen ist schließlich für die Akzeptanz der AOD der einfache Zugriff auf aspektspezifische Daten notwendig. Dieser umfasst auf der einen Seite die Integration und Verwaltung funktionaler Aspekte und ihrer Metadaten in bestehenden Datenmodellen durch den Anwendungs-Administrator, welcher durch ein geeignetes Werkzeug wie in [Dit11] beschrieben unterstützt werden sollte. Darüber hinaus bedarf es flexibler Schnittstellen zur anwendungsseitigen Abfrage, Verarbeitung und Manipulation der fachlichen Daten unter Aspekteinfluss.

3.4 Analyse von Persistierungsmodellen

Nach Formulierung des AOD-Paradigmas in Abschnitt 3.3 sollen nun Möglichkeiten zur Realisierung der damit verbundenen Anforderungen im Kontext verschiedener Persistierungsmodelle untersucht werden. Konkret sind jene durch Abschnitt 3.1.2.1 aufgezeigten inhaltlichen Auswirkungen funktionaler Aspekte mit den verfügbaren Konstrukten des jeweiligen Persistierungsmodells zu implementieren, wobei insbesondere die Unterstützung des Hyperquader-Modells gemäß Abbildung 3.2 im Vordergrund steht. Die Auswahl der Persistierungsmodelle erfolgt primär aus Sicht der praktischen Nutzbarkeit in bestehenden Infrastrukturen, weswegen zwei klassische DBMS in Abschnitt 3.4.1 und Abschnitt 3.4.2 sowie als etablierte Alternative XML in Abschnitt 3.4.3 nachfolgend analysiert und schließlich in Abschnitt 3.5 bewertet werden.

3.4.1 RDBMS

Ein Relationales Datenbankmanagementsystem (RDBMS) dient der Verwaltung und Speicherung von Daten basierend auf dem relationalen Modell [Cod70], wobei die Ursprünge unter anderem in der Entwicklung von *System R* durch IBM Mitte der 1970er Jahren liegen. In diesem Zusammenhang wurde mit SEQUEL auch eine Möglichkeit für den Zugriff auf relationale Daten geschaffen [CB74], aus welcher sich einige Jahre später SQL als Abfragesprache entwickelt hat. Deren einfach zu erlernende Benutzung und erfolgreiche Normierung haben wesentlich dazu beigetragen, dass RDBMS heutzutage die mit Abstand am häufigsten in der Praxis eingesetzte Datenbank-Technologie darstellen. Entsprechende Anwendungen bieten unter anderem Software-Hersteller wie IBM, Oracle oder Microsoft an, zudem gibt es etliche Open-Source Produkte wie beispielsweise MySQL.

3.4.1.1 Technologie und Begriffe

Das grundlegende Konstrukt in RDBMS ist die Relation zur Speicherung von Datensätzen in Form von Tupeln, deren Struktur jeweils durch die Attribute des Relationenschemas bestimmt wird und wie in **Abbildung 3.6** visualisiert werden kann.

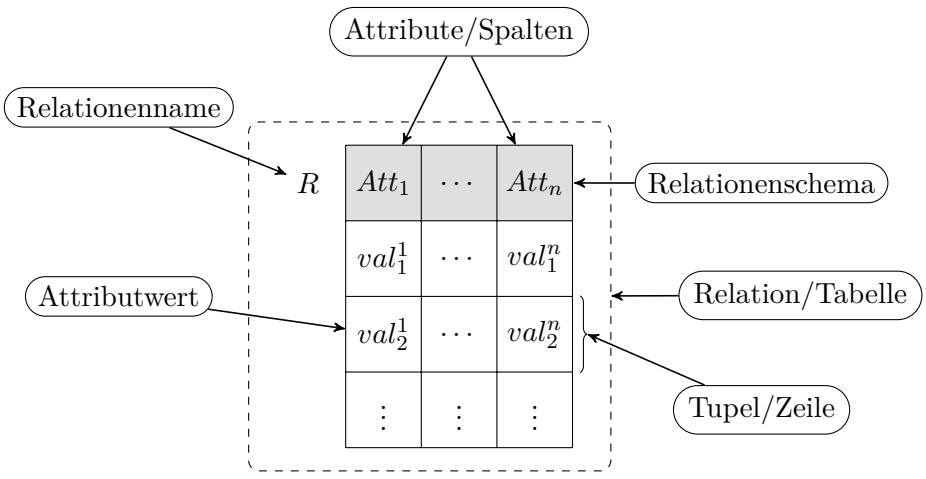


Abbildung 3.6: Begriffe in relationalen Datenbanken

Dabei unterstützen Attribute gemäß der ersten Normalform nur skalare Datentypen wie Zahlen oder Zeichen, womit Attributwerte nicht strukturierbar sind [Cod70]. Darüber hinaus können vielfältige Integritätsbedingungen definiert werden. Eine besondere Rolle spielt hierbei die Auszeichnung einer Attributmenge als sogenannter Primärschlüssel der Relation zur Kennzeichnung der Werteindeutigkeit über alle Tupel sowie der funktionalen Abhängigkeit der restlichen Attribute von jenem Schlüssel. Zudem lassen sich Relationen untereinander über sogenannte Fremdschlüssel in Beziehung setzen. Für weitergehende Details und Erläuterungen sei an dieser Stelle auf [HR01] verwiesen.

3.4.1.2 Diskussion von Lösungskonzepten

Ausgehend von dem in Abschnitt 3.1.2 geschilderten Einfluss funktionaler Aspekte lässt sich dieser aus Sicht der Attribute auch als zusätzliche Abhängigkeit ausdrücken. Der naheliegende Gedanke, diese Abhängigkeiten durch Erweiterung des Primärschlüssels einer Relation zu modellieren, ist in **Abbildung 3.7** für das Anwendungsbeispiel aus Abschnitt 3.2 skizziert. Die dadurch gewonnene Nähe zum Star-Schema [KR02] bringt allerdings auch dessen bereits in Abschnitt 2.5.4.1 identifizierten Probleme mit sich. Unter anderem ergibt sich hierbei für alle Attribute der Relation Module die gemeinsame Abhängigkeit von den gleichen funktionalen Aspekten. Eine individuelle Definition relevanter Aspekte je Attribut entsprechend des in Abbildung 3.2 dargestellten Hyperquader-Modells ist nicht möglich. Zudem sind die Anforderungen der Modularität und Lokalität nicht erfüllt.

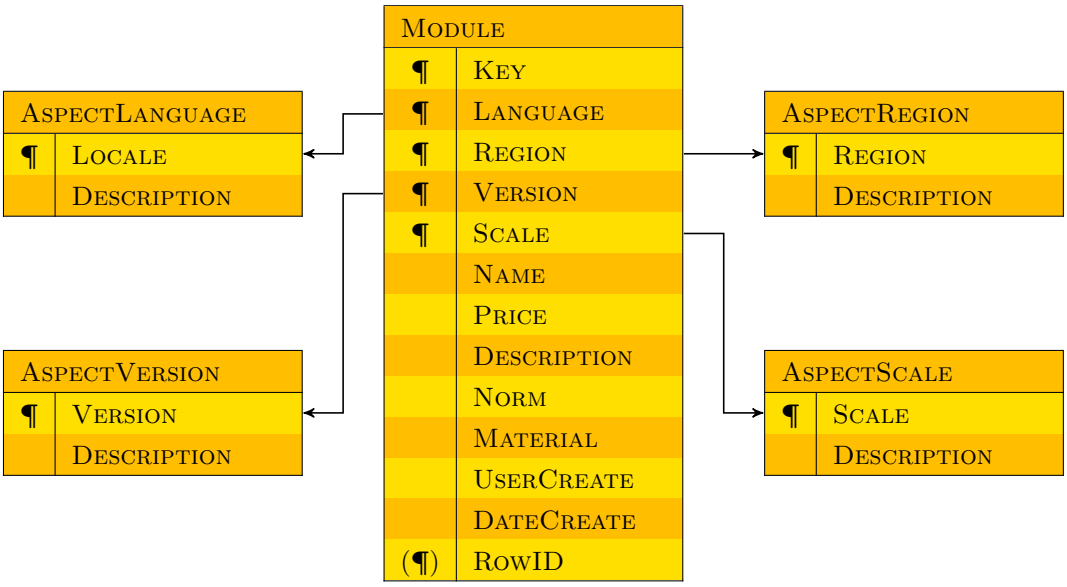


Abbildung 3.7: Unterstützung funktionaler Aspekte durch Primärschlüssel-Erweiterung

Eine Übersicht in [Sch11] zu verschiedenen Realisierungstechniken im relationalen Modell zeigt, dass die gleichzeitige Gewährleistung aller Anforderungen des AOD-Paradigmas gemäß Abschnitt 3.3 nur sinnvoll durch Auslagerung der aspektspezifischen Werte aus den fachlichen Tabellen erreicht werden kann. Aufgrund der geforderten Universalität muss der Bezug aus diesen separaten Relationen zum jeweils ursprünglichen Attributwert auf generische Art und Weise erfolgen, wobei zur eindeutigen Bestimmung eines Attributwertes im relationalen Modell das Tripel (Tabelle, Zeile, Spalte) genügt. Ein solches Tripel

stellt auch die Grundlage für das sogenannte Entity-Attribute-Value (EAV)-Konzept dar [NMC+99], welches die Zuordnung eines Wertes zu einem beliebigen Attribut einer Entität, identifiziert durch Tabelle und Zeile, in nur einer Relation ermöglicht. Die Nutzung dieses Prinzips zur Persistierung jeglicher aspektspezifischer Attributwerte ist für das Anwendungsbeispiel ansatzweise in **Abbildung 3.8** illustriert.

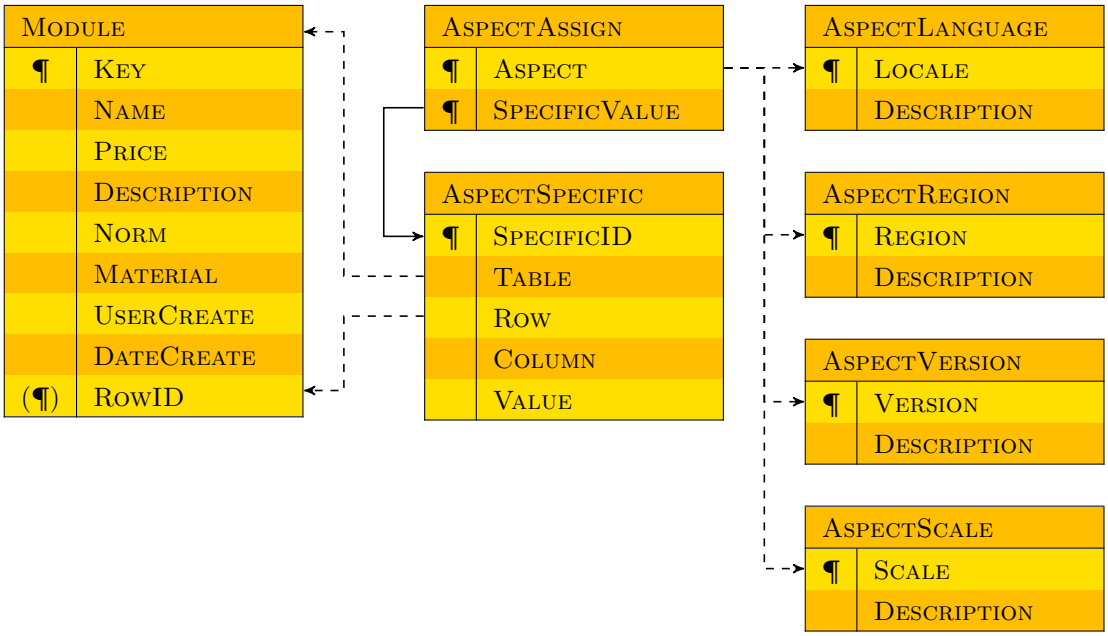


Abbildung 3.8: Unterstützung funktionaler Aspekte durch das EAV-Konzept

Während die Relation AspectSpecific zur Speicherung eben jener aspektspezifischen Werte dient, wird die eigentliche Zuordnung dieser Werte zur Menge der beeinflussenden Aspekte in der Relation AspectAssign hinterlegt. Dadurch ist es möglich, die Abhängigkeit eines Attributs und damit seiner aspektspezifischen Werte von mehreren Aspekten abzubilden wie in **Abbildung 3.9** dargestellt. Beispielsweise ist dabei gemäß Abschnitt 3.2.2 der Wert '0.47' für das Attribut Module.Price unter den Aspekten Region (=EU), Version (=2) und Staffel (=Standard) gültig. Auf die Infrastruktur für zusätzliche Metadaten entsprechend [Lie10b] wurde hier aus Übersichtsgründen verzichtet. Eine vollständige Beschreibung dieses Ansatzes sowie des EAV-Konzepts findet in Abschnitt 4.3 statt.

Aspect	SpecificValue
de	1001
1	1001
en	1002
1	1002
EU	1003
2	1003
Standard	1003

SpecificID	Table	Row	Column	Value
1001	Module	3141	Name	Schraube
1002	Module	3141	Name	screw
1003	Module	3141	Price	0.47

Abbildung 3.9: Beispieldaten für AspectAssign und AspectSpecific (RDBMS)

3.4.2 ORDBMS

Ein Objektrelationales Datenbankmanagementsystem (ORDBMS) lässt sich beschreiben als ein um objektorientierte Konzepte erweitertes RDBMS. Gründe für diese ab Mitte der 1990er Jahre entstandene „Weiterentwicklung“ waren die Forderung, auch zunehmend komplexer strukturierte Daten, beispielsweise aus dem Ingenieursbereich, in Datenbanken speichern zu können sowie der zeitgleich beginnende Siegeszug von OOP verbunden mit der Notwendigkeit, objektorientierte Anwendungsdaten persistieren zu müssen. In Bezug auf RDBMS zeigt sich dabei jedoch der sogenannte *Impedance Mismatch*, welcher unter anderem durch den Verlust von logischer Zusammengehörigkeit der Objekte, deren nicht-intuitive Abbildung auf relationale Strukturen der Datenbank und Performance-Einbußen aufgrund komplexerer Verbund-Operationen gekennzeichnet ist [Ska05, Rus08]. Da die Entwicklung reiner objektorientierter Datenbankmanagementsysteme in der Praxis vor allem wegen der fehlenden Leistungsfähigkeit und Kompatibilität zu Anwendungsschnittstellen wie Open Database Connectivity (ODBC) nicht den erhofften Durchbruch erzielt hat, stellen ORDBMS eher die evolutionäre Weiterentwicklung bestehender RDBMS sowie der Zugriffssprache SQL dar. Heutzutage zählen die Datenbank-Produkte von Herstellern wie IBM und Oracle sowie das Open-Source-Projekt PostgreSQL zu den ORDBMS mit weitestgehender Unterstützung der SQL:1999-Norm [Tür03].

3.4.2.1 Technologie und Begriffe

Entsprechend der zuvor geschilderten Motivation von ORDBMS besitzen diese prinzipiell alle auch bei RDBMS vorhandenen Konstrukte. Darüber hinaus äußert sich der objektorientierte Einfluss durch die SQL:1999-Norm insbesondere anhand von Vererbungsmechanismen, der Referenzierung eines Tupels über den systeminhärenten Objektidentifikator (OID) und nutzerdefinierbarer sowie komplexer vordefinierter Datentypen. Letztere sind in **Abbildung 3.10** schematisch dargestellt, wobei zur Unterstützung des Hyperquader-Modells aus Abbildung 3.2 der Tupeltyp (ROW) als Ansammlung heterogener Elemente und der Kollektionstyp (ARRAY) als geordnete Ansammlung homogener Elemente auffallen. Allerdings ist durch die SQL:1999-Norm nur deren einfache Verschachtelung zum Subtabellentyp erlaubt. Diese Beschränkung der Schachtelbarkeit und -tiefe wurde mit Verabschiedung der SQL:2003-Norm aufgehoben, welche gleichzeitig auch mit einem Tabellentyp (MULTISET) den Kollektionstyp erweiterte [TS05, Tür03]. Allerdings weichen diesbezüglich typischerweise reale ORDBMS-Produkte von der Norm ab.

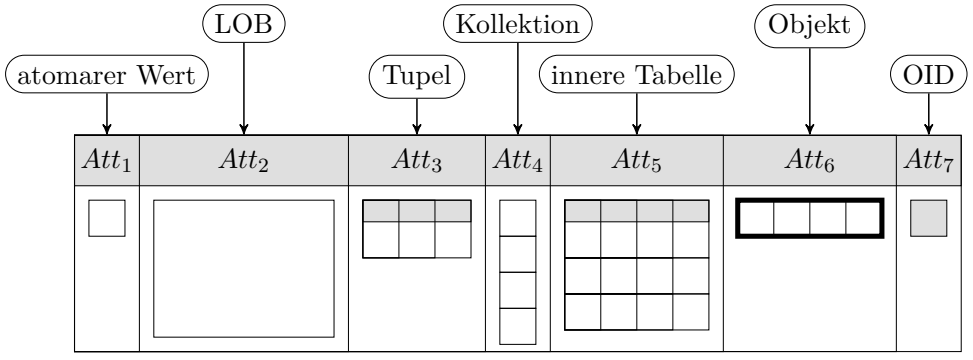


Abbildung 3.10: Attributtypen in ORDBMS nach [TS05]

3.4.2.2 Diskussion von Lösungskonzepten

Die Möglichkeit, Attribute entgegen der ersten Normalform (Stichwort NF2) mit komplexen Datentypen zu definieren, könnte prinzipiell genutzt werden, um entweder bereits beim Entwurf des Datenmodells die potentielle Abhängigkeit von beliebig vielen funktionalen Aspekten zu berücksichtigen oder die für einen konkreten Aspekt relevanten Attribute zum Zeitpunkt der Integration anzupassen. In beiden Fällen würden sowohl die Eigenschaften der Modularität als auch der Lokalität des AOD-Paradigmas verloren gehen. Daher wird im Folgenden die Auslagerung von aspektspezifischen Daten in eigene Relationen betrachtet. Analog zur Argumentation bei RDBMS kommen dafür auch hier die generischen Strukturen des EAV-Konzepts zum Einsatz. Allerdings kann unter Nutzung der komplexen Datentypen nun zu einem über das Tripel (Tabelle, Zeile, Spalte) referenzierten Attributwert der Originaltabelle nicht nur ein Wert, sondern die Menge aller aspektspezifischen Ausprägungen hinterlegt werden. Aufgrund der Tatsache, dass ein fachliches Attribut von mehreren Aspekten (Aspektmenge) abhängig sein kann, ergibt sich die in **Abbildung 3.11** skizzierte Struktur für das Attribut AspectSpecific.AspectValues im Rahmen des Anwendungsbeispiels aus Abschnitt 3.2.

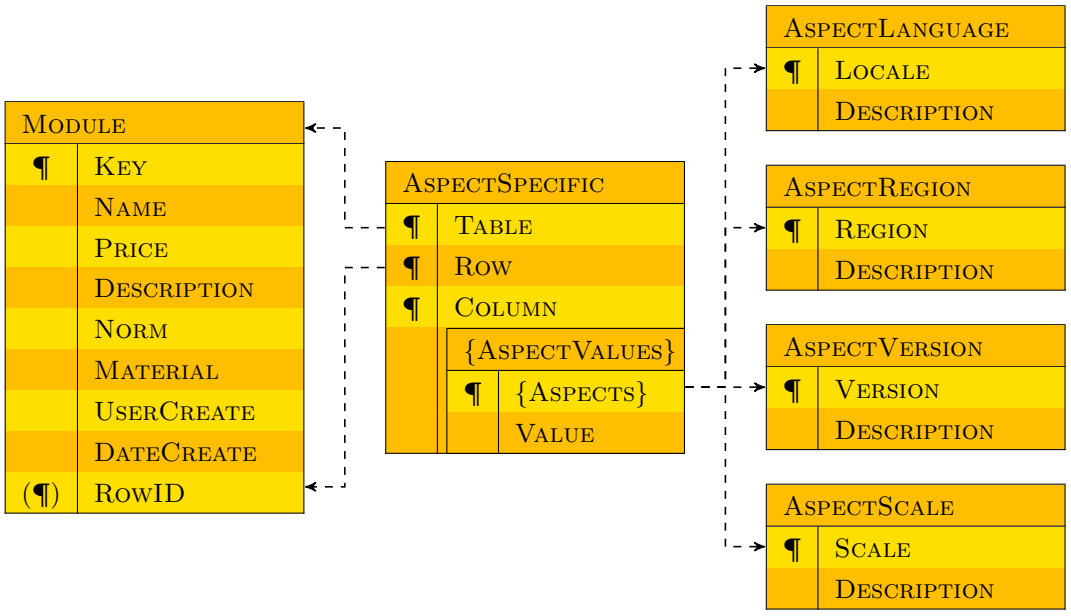


Abbildung 3.11: Unterstützung funktionaler Aspekte durch ROW- und ARRAY-Typen

In **Abbildung 3.12** wird nicht nur die Funktionsweise der verschachtelten Datentypen in AspectSpecific offensichtlich, es zeigen sich auch basierend auf den analog Beispieldaten gegenüber der relationalen Modellierung in Abbildung 3.8 zwei große Unterschiede.

Table	Row	Column	{AspectValues}	
			{Aspects}	Value
Module	3141	Name	[{de, 1}, Schraube], [{en, 1}, screw]	
Module	3141	Price	[{EU, 2, Standard}, 0.47]	

Abbildung 3.12: Beispieldaten für AspectSpecific (ORDBMS)

Einerseits werden nur zwei komplexe statt sieben einfache Tupel benötigt, um die gleiche Menge aspektspezifischer Daten zu speichern. Andererseits tritt durch die mengenwertige Notation {...} innerhalb eines Tupels [...] kein Informationsverlust hinsichtlich der Gruppierung von Daten wie in Abbildung 3.9 auf. Allerdings besteht auch hier für das Attribut AspectSpecific.AspectValues.Value die Anforderung zur Unterstützung verschiedener Datentypen, zudem ist die Verwaltung von Metadaten zu Tabellen, Spalten und Aspektschlüsseln essentiell. Schließlich wäre eine weitere Verschachtelung für alle Spalten einer Zeile und alle Zeilen einer Tabelle wie in [Ger12] beschrieben möglich.

3.4.3 XML

Die Extensible Markup Language (XML) ist eine Meta-Auszeichnungssprache, welche zur Beschreibung von hierarchischen Datenstrukturen eingesetzt werden kann und basierend auf der Standard Generalized Markup Language (SGML) im Jahr 1998 als offizielle Empfehlung vom World Wide Web Consortium (W3C) verabschiedet wurde [Mül08]. Seitdem hat sich XML als maschinenlesbares und plattformunabhängiges Austauschformat zwischen Computerverbünden wie dem Internet etabliert, wodurch sich ab der Jahrtausendwende der Bedarf zur Speicherung von XML-Daten sowie deren integrierten Auswertung mittels geeigneter Zugriffssprachen wie XPath oder XQuery [W3C99, W3C07] in Datenbanken herauskristallisierte. Dies führte sowohl zur Beachtung der XML-Thematik in der SQL:2003-Norm [ISO03] als auch zur Erweiterung bereits existierender RDBMS-Produkte um Funktionalität für die Verarbeitung und Abbildung von XML. Als Alternative entstanden zudem sogenannte „native“ XML-Datenbanken zur direkten Speicherung und Prozessierung von XML-Dokumenten. Trotz einiger am Markt recht erfolgreicher Produkte, wie zum Beispiel der Tamino XML Server[2], spielen diese gegenüber „XML-enabled“ RDBMS bisher eine eher untergeordnete Rolle und werden hier nicht weiter betrachtet.

3.4.3.1 Technologie und Begriffe

Basierend auf [Von11, Mül08] und anhand des in **Abbildung 3.13** dargestellten Beispiels soll nachfolgend ein kurzer Überblick zu den wichtigsten Bestandteilen und dem generellen Aufbau von XML-Dokumenten gegeben werden.

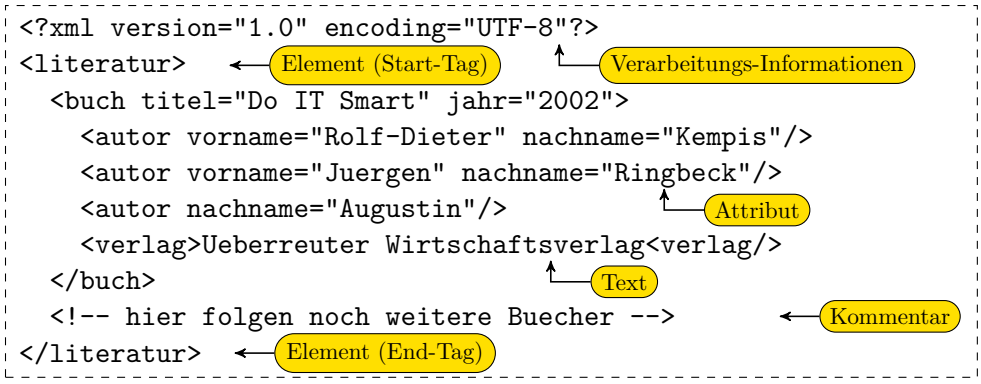


Abbildung 3.13: Beispiel eines XML-Dokuments

[2] http://www.softwareag.com

Die zentrale Struktureinheit eines XML-Dokuments ist das Element, welches weitere (Kind-)Elemente oder Text zwischen einem gleichnamigen Start-Tag `<element>` und End-Tag `</element>` enthalten kann. Inhaltslose Elemente wie `<autor/>` in Abbildung 3.13 können in abkürzender Schreibweise notiert werden. Weiterhin lassen sich zu Elementen beliebige Attribute eines skalaren Typs definieren und Kommentare durch eine spezielle Kennzeichnung platzieren. XML ermöglicht semistrukturierte Daten, da die Struktur eines XML-Dokuments nicht zwingend vorgegeben sein muss und somit durch die Daten selbst beschreibbar ist. Werden dabei alle in [W3C98] formulierten Syntaxregeln der XML-Spezifikation strikt eingehalten, spricht man von wohlgeformtem XML. Dies umfasst unter anderem die Existenz genau eines Wurzel-Elements, ein schließendes End-Tag zu jedem Start-Tag eines Elements sowie deren ebenentreue Verschachtelung. Darüber hinaus lässt sich die Struktur eines XML-Dokuments anwendungsspezifisch durch Zuordnung eines XML-Schemas [W3C01] einschränken. Genügt ein wohlgeformtes XML-Dokument diesem Schema, dann wird es auch als valide bezeichnet.

3.4.3.2 Diskussion von Lösungskonzepten

Eine prägende Eigenschaft des zu unterstützenden Hyperquader-Modells gemäß Abbildung 3.2 sind die verschiedenen Dimensionen je Attribut abhängig von den jeweils wirkenden funktionalen Aspekten. Während diese Flexibilität mit den bisher betrachteten strukturierten Datenmodellen von RDBMS und ORDBMS nur schlecht realisierbar ist, eignet sich XML für diese Aufgabe aufgrund der inhärenten Semistrukturiertheit bei erster Betrachtung deutlich besser. Im Folgenden wird von einem fachlichen Datenmodell in einem „XML-enabled“ RDBMS ausgegangen mit dem Ziel, die aspektspezifischen Daten in Attributen vom Typ XML zu speichern. Analog zur Argumentation bei ORDBMS müssen diese aspektspezifischen Attribute jedoch außerhalb der fachlichen Relationen definiert sein, um die Lokalität und Modularität gewährleisten zu können. Dadurch bietet sich zur generischen Verknüpfung von aspektspezifischen und fachlichen Werten erneut das EAV-Konzept an, wie dies **Abbildung 3.14** für das Anwendungsbeispiel aus Abschnitt 3.2 illustriert. Die XML-Struktur am Attribut AspectSpecific.Value unterstützt dabei eine Menge von aspektspezifischen Werten (`<SpecificValue>`) mit jeweils einer Menge von Aspektschlüsselwerten (`<Aspect>`).

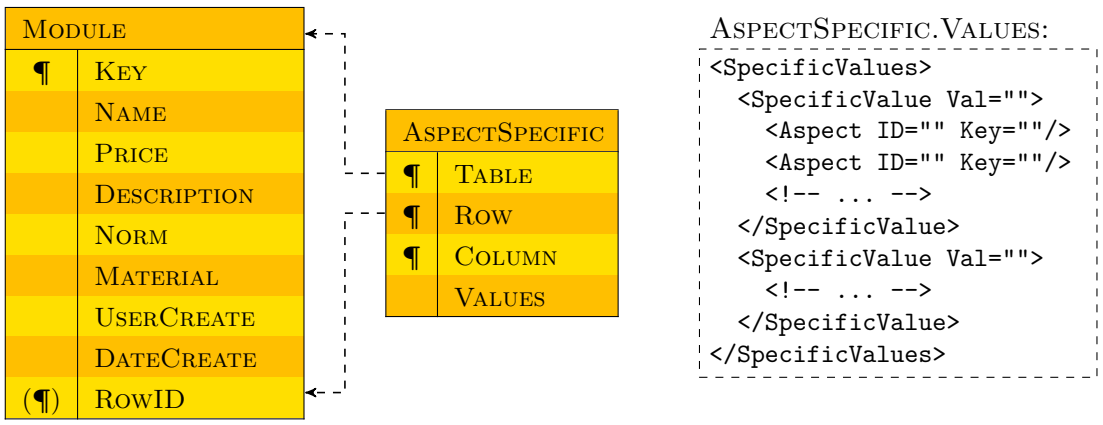


Abbildung 3.14: Unterstützung funktionaler Aspekte mit dem XML-Datentyp

Die Übertragung der Beispieldaten aus Abbildung 3.9 für den soeben erläuterten Ansatz ergibt die in **Abbildung 3.15** dargestellte Befüllung der Tabelle AspectSpecific. Dabei wird in jedem Attributwert von AspectSpecific.Values ein XML-Dokument gespeichert, welches gemäß der in Abbildung 3.14 definierten Struktur alle aspektspezifischen Ausprägungen des über (Table, Row, Column) referenzierten Wertes enthält.

Diese Form der Realisierung stellt allerdings nur eine mögliche Variante zur XML-Nutzung dar und erleichtert insbesondere die Vergleichbarkeit mit den bereits untersuchten Persistierungsformen. Alternativ könnten auch die Angaben zum referenzierten Attributwert im XML-Dokument hinterlegt werden. Wie in [Ger12] beschrieben, würde im Extremfall eine Relation mit einem einzigen Attribut vom Typ XML ausreichen, um darin alle aspektspezifischen Werte jeglicher Attribute aus den fachlichen Tabellen zu speichern. Aufgrund der Verarbeitungs-Charakteristik von XPath bzw. XQuery ist diese Variante jedoch nicht favorisiert, auf die genauen Hintergründe wird in Abschnitt 3.5.2 im Rahmen der entsprechenden Bewertung eingegangen.

Table	Row	Column	Values

Module	3141	Name	`<?xml version="1.0" encoding="UTF-8"?>`
			`<SpecificValues>`
			`  <SpecificValue Value="Schraube">`
			`  <Aspect ID="Language" Key="de"/>`
			`  <Aspect ID="Version" Key="1"/>`
			`  </SpecificValue>`
			`  <SpecificValue Value="screw">`
			`  <Aspect ID="Language" Key="en"/>`
			`  <Aspect ID="Version" Key="1"/>`
			`  </SpecificValue>`
			`</SpecificValues>`

Module	3141	Price	`<?xml version="1.0" encoding="UTF-8"?>`
			`<SpecificValues>`
			`  <SpecificValue Value="0.47">`
			`  <Aspect ID="Region" Key="EU"/>`
			`  <Aspect ID="Version" Key="2"/>`
			`  <Aspect ID="Scale" Key="Standard"/>`
			`  </SpecificValue>`
			`</SpecificValues>`

Abbildung 3.15: Beispieldaten für AspectSpecific (XML)

3.5 Bewertung der Persistierungsmodelle

Dieser Abschnitt dient der abschließenden qualitativen Betrachtung aller in Abschnitt 3.4 diskutierten Persistierungsmodelle hinsichtlich der Unterstützung von funktionalen Aspekten. Dafür werden zunächst in Abschnitt 3.5.1 geeignete Kriterien definiert, anhand derer in Abschnitt 3.5.2 der Vergleich und die Bewertung stattfindet. Daraus resultierende Erkenntnisse werden schließlich in Abschnitt 3.5.3 als Fazit und Überleitung zum Kapitel 4 zusammengefasst.

3.5.1 Kriterien

Für die objektive Bewertung der Persistierungsmodelle werden basierend auf [Sch11, Ger12] die Anforderungen des AOD-Paradigmas, die Praxisrelevanz, die Skalierbarkeit, die Performance sowie die Speichereffizienz als Maßstab zu Grunde gelegt und zu diesem Zweck nachfolgend erläutert.

AOD-Paradigma: Wie in Abschnitt 3.3 definiert, sind mit dem AOD-Paradigma die Eigenschaften der Modularität, Orthogonalität, Lokalität, Universalität und Benutzerfreundlichkeit verbunden. Demzufolge ist deren Gewährleistung für ein konkretes Persistierungsmodell zu prüfen. Dabei soll insbesondere bezüglich der Benutzbarkeit abgeschätzt werden, welcher Aufwand sich zur Administration funktionaler Aspekte und für die Verarbeitung aspektspezifischer Daten ohne zusätzliche Anwendungslogik ergibt.

Praxisrelevanz: Dieses Kriterium erlaubt für ein Persistierungsmodell die Bewertung hinsichtlich seiner praktischen Verwendbarkeit, welche vor allem vom Verbreitungsgrad sowie der zuvor betrachteten Benutzerfreundlichkeit direkt beeinflusst wird. Durch die Konzeption des AOD-Paradigmas als Antwort auf die in Abschnitt 2.5.5 formulierten Herausforderungen bezüglich der Datenhaltung (nicht nur) in Produktkonfiguratoren wird dem Kriterium der Praxisrelevanz folglich ein besonderer Stellenwert beigemessen.

Skalierbarkeit: Im Sinne einer Erweiterung der im AOD-Paradigma genannten Lokalitäts-Anforderung dient das Merkmal der Skalierbarkeit zur Bewertung eines Persistierungsmodells hinsichtlich seiner Flexibilität bei wachsenden Mengen von funktionalen Aspekten, deren Ausprägungen sowie den aspektspezifischen Daten. Insbesondere ist von Interesse, wie generisch die Infrastruktur zur Verwaltung dieser Informationen ist.

Performance: Ausgehend von der zuvor betrachteten Skalierbarkeit spielt auch die Performance der Persistierungsmodelle eine entscheidende Rolle. Dabei sind auch im Kontext funktionaler Aspekte die Anzahl der zu verknüpfenden Relationen und deren Kardinalität sowie die Verfügbarkeit von Indexstrukturen maßgebliche Einflussfaktoren für eine Abschätzung von Zugriffsoperationen, wie beispielsweise das Lesen oder Verändern eines aspektspezifischen Wertes zu einem Attributwert einer fachlichen Tabelle.

Speichereffizienz: Schließlich soll mit Hilfe der Speichereffizienz beurteilt werden, welches Verhältnis zwischen den reinen aspektspezifischen Daten und den im jeweiligen Persistierungsmodell notwendigen Verwaltungsdaten zur Speicherung und Zuordnung entsteht. Dieses Kriterium hat wiederum auch direkten Einfluss auf die Performance, da kompakt gespeicherte Informationen schneller transportiert und verarbeitet werden können. Insbesondere ist zu prüfen, ob Daten in irgendeiner Form redundant speicherbar sind, was den Wartungsaufwand erhöhen und somit die Benutzerfreundlichkeit reduzieren kann.

3.5.2 Vergleich

Zur Bewertung der Persistierungsmodelle RDBMS, ORDBMS und XML werden diese anhand jeweils eines in Abschnitt 3.5.1 aufgeführten Kriteriums gemeinsam betrachtet und miteinander verglichen. Eine zusammenfassende Darstellung der Bewertungsergebnisse findet sich nachfolgend in **Tabelle 3.1**.

Kriterium		RDBMS	ORDBMS	XML
AOD-Paradigma	Modularität	++	++	++
	Orthogonalität	++	++	++
	Lokalität	+	+	+
	Universalität	++	+	+
	Benutzerfreundlichkeit	–	–	––
Praxisrelevanz		++	–	+
Skalierbarkeit		+	+	+
Performance		–	+	–
Speichereffizienz		++	++	–

++: sehr gut +: gut –: schlecht ––: sehr schlecht

Tabelle 3.1: Bewertung der Persistierungsmodelle

Die Anforderung der Modularität des **AOD-Paradigmas** wird von allen drei Persistierungsmodellen vollständig erfüllt aufgrund der Speicherung der Metadaten zu Aspekten in eigenen Relationen (z.B. AspectLanguage) und der Auslagerung aspektspezifischer Daten in die unabhängige Relation AspectSpecific bzw. AspectAssign im Fall von RDBMS. Ebenso wird auch die Eigenschaft der Orthogonalität ohne Ausnahme erfüllt. Einerseits stellen sich Abhängigkeiten zwischen Aspekten und Attributen nur als prinzipiell ungeordnete Datenmengen dar, nämlich konkret

- für RDBMS durch Tupel in den Relationen AspectSpecific und AspectAssign,
- für ORDBMS durch Tupel in der Relation AspectSpecific oder Elemente in AspectSpecific.AspectValues.{Aspects} sowie
- für XML durch Tupel in der Relation AspectSpecific oder `Aspect`-Elemente im XML-Dokument.

Andererseits ist durch die Modularität der aspektspezifischen Relationen die gegenseitige Beeinflussung von abhängigen Attributen ausgeschlossen. Basierend auf dieser Argumentation ist auch die Lokalität gegeben, allerdings muss für das zugrunde liegende EAV-Konzept die Existenz eines ein-elementigen ganzzahligen Schlüssels analog zu Module.RowID gewährleistet sein. Dieser kann jedoch prinzipiell einer Relation bei Bedarf hinzugefügt werden. Nach der bisherigen Gleichförmigkeit in der Bewertung treten bei Betrachtung der Universalität erste Unterschiede zwischen den Persistierungsmodellen auf. Während der Lösungsansatz für RDBMS vollständig zur SQL:92-Norm kompatibel ist, benötigen die Ansätze für ORDBMS und XML zwingend die Konstrukte der SQL:2003-Norm. Dabei kann es in der praktischen Nutzung zu Einschränkungen kommen, die im zugehörigen Kriterium aufgegriffen werden. Als letzte Anforderung des AOD-Paradigmas

muss die Benutzerfreundlichkeit bei der Anwendungsentwicklung für alle Persistierungsmodelle als schlecht eingestuft werden, da mit dem in der Relation AspectSpecific verwendeten EAV-Konzept die Anfragekomplexität steigt wie nachfolgend in Kapitel 4 erläutert. Zudem wird im Fall von XML noch eine weitere Anfragesprache notwendig, um die Informationen im XML-Dokument auswerten zu können.

Wie bei Beurteilung der Universalität angedeutet, hat diese direkten Einfluss auf die Einschätzung der **Praxisrelevanz** aufgrund der Unterstützung bestimmter SQL-Normen in den Datenbank-Produkten. Hierbei gilt: je älter eine Normierung, desto größer ist deren Präsenz in den Produkten. Dadurch besitzen RDBMS wie bereits in Abschnitt 3.4.1 geschildert eine hohe Bedeutung und Verbreitung. Diese ist für ORDBMS demzufolge deutlich schlechter, da jene Technologie erst traditionelle Anwendungssysteme auf Basis von RDBMS verdrängen müsste. Zudem setzen unter anderem Oracle und DB2 die SQL:2003-Norm in Bezug auf die komplexen Datentypen sehr unterschiedlich um [TS05]. Dies gilt auch für die direkte Unterstützung von XML in RDBMS [Mül08], allerdings hat XML aufgrund seiner erläuterten Einsatzmöglichkeiten (siehe Abschnitt 3.4.3) darüber hinaus eine höhere praktische Bedeutung.

Alle drei Persistierungsmodelle bieten eine gute **Skalierbarkeit**. Zwar steigt die Anzahl der Relationen in den präsentierten Lösungsansätzen mit jedem funktionalen Aspekt. Durch eine generalisierte Infrastruktur, wie sie im Kapitel 4 vorgestellt wird, kann jedoch eine konstante Menge an Metadaten-Relationen geschaffen werden. Dagegen ist für die Relation AspectSpecific mit einem dauerhaften Anstieg der Kardinalität zu rechnen, dem bei Bedarf durch Anwendung typischer Partitionierungsstrategien begegnet werden sollte.

Die schon zur Bewertung der Benutzerfreundlichkeit identifizierte Komplexität des EAV-Konzepts bringt nun auch für die **Performance** gewisse Einschränkungen mit. Im Fall von RDBMS ist gegenüber den anderen beiden Persistierungsmodellen noch mindestens eine weitere Verbundoperation zur Relation AspectAssign notwendig, welche sich insbesondere aufgrund der Kardinalitäts-Abschätzungen beim Kriterium der Skalierbarkeit negativ auf die Anfrageausführung auswirkt. Diese sollte demnach für den Ansatz im ORDBMS schneller möglich sein, sofern die Verarbeitung komplexer Typen vom System effizient durchgeführt wird. Durch die Auswertung und Integration unterschiedlicher Sprachen (SQL, XQuery/XPath) ist wiederum für den XML-Ansatz mit einer etwas schlechteren Performance zu rechnen.

Schließlich sind bezüglich der **Speichereffizienz** die beiden Ansätze in den Persistierungmodellen RDBMS und ORDBMS als sehr kompakt und speichersparend einzuordnen. Dagegen verursacht die Semistrukturiertheit von XML einen gewissen Overhead an Metainformationen gegenüber den Nutzdaten, was zu einer schlechteren Bewertung gegenüber den beiden zuvor betrachteten Ansätzen führt.

3.5.3 Fazit

Als erstes Ergebnis des Vergleichs in Abschnitt 3.5.2 lässt sich festhalten, dass die vorgestellten Persistierungsmodelle für RDBMS, ORDBMS und XML alle in gleichem Maße die Anforderungen des AOD-Paradigmas erfüllen und damit prinzipiell die Unterstützung funktionaler Aspekte ermöglichen. Allerdings ergeben sich für die weiteren Bewertungskriterien gemäß Tabelle 3.1 teilweise deutliche Unterschiede, wobei im ungewichteten Ver-

gleich XML gegenüber den beiden Alternativen am wenigsten geeignet erscheint. Wird noch die enorme Bedeutung des Kriteriums der Praxisrelevanz in den Entscheidungsprozess einbezogen, stellt sich das Persistierungsmodell auf Basis von RDBMS letztlich als die sinnvollste Variante zur Realisierung des AOD-Paradigmas dar.

Diese Tendenz wird noch dadurch verstärkt, dass die AOD insbesondere die Verwaltung funktionaler Aspekte für bestehende Anwendungen unterstützen soll, deren Persistierungsebene aufgrund der historischen Entwicklung in den meisten Fällen auf dem relationalen Modell beruht. Dazu kommt, dass die ausgelagerte Speicherung der aspektspezifischen Daten in einem Spezialsystem neben dem RDBMS für die fachlichen Daten zusätzliche Schnittstellen und Konsistenzprüfungen erzwingen würde und dadurch sowohl aus technischer als auch praxisbezogener Sicht keine Lösung darstellt.

Zusammenfassend erscheint es zielführend, den weiteren Verlauf der Arbeit auf die Persistierung mit RDBMS einzuschränken. Aus diesem Grund wird im anschließenden Kapitel 4 basierend auf den Ansätzen in Abschnitt 3.4.1 ein relationales Referenzmodell zur Abbildung funktionaler Aspekte und Unterstützung des AOD-Paradigmas vorgestellt.

Kapitel 4

Relationales Referenzmodell

Dieses Kapitel dient der Beschreibung eines Referenzmodells für die AOD, welches unter Berücksichtigung der vorangegangenen Bewertung primär Konstrukte und Konzepte von RDBMS nutzt. Hierzu findet einleitend in **Abschnitt 4.1** eine Abgrenzung und Fokussierung des Referenzmodells hinsichtlich des thematischen Kontexts dieser Arbeit statt. Darauf aufbauend erfolgt in **Abschnitt 4.2** eine detaillierte und formalisierte Betrachtung aspektspezifischer Daten im relationalen Modell. Die dabei gewonnenen Erkenntnisse bilden die Grundlage für das eigentliche Persistenzmodell, dessen Strukturen und Wirkungsweise zur Verwaltung aspektabhängiger Daten in **Abschnitt 4.3** vorgestellt werden. Zur praktischen Nutzung und Anwendungsintegration des Referenzmodells ist schließlich auch ein Zugriffsmodell notwendig, welches durch **Abschnitt 4.4** näher erläutert wird.

4.1 Abgrenzung und Zielsetzung

Während den Auswirkungen querschnittlicher Belange im Programmcode durch das Konzept der AOP [KLM+97] umfassend und anwendungsübergreifend begegnet werden kann, fehlt bisher ein ähnlich generischer Ansatz für die Datenebene. Stattdessen ist zu beobachten, dass einerseits die verschiedenen Arbeiten in diesem Umfeld deutlich vom jeweiligen Anwendungsfall beeinflusst werden und andererseits eine Fokussierung auf bestimmte Teilaspekte erfolgt. Stellvertretend seien hierbei die Aspektorientierten Datenbanksysteme zur Verwaltung von AOP-beeinflusstem Code [Ras04] sowie konzeptuelle Erweiterungen der Relationenalgebra für aspektabhängige Tupel [Dyr11] erwähnt. Darüber hinaus weisen die in [BCTV04] vorgestellten Annotationen und deren Verarbeitung gewisse Ähnlichkeit mit den in Abschnitt 3.1 eingeführten funktionalen Aspekten auf, wobei die verwendete Abbildung mittels zusätzlicher Attribute in den fachlichen Relationen gegen die hier geforderte Modularität des AOD-Paradigmas verstößt.

Das im Folgenden vorgestellte Referenzmodell umfasst sowohl die Speicherung als auch die Auswertung von funktionalen Aspekten in RDBMS. Hierbei findet jedoch eine Beschränkung auf aspektspezifische *Attributwerte* statt, d.h. gemäß Klassifikation in Abschnitt 3.1.2 sollen nur Änderungen zu Attributen für existierende Tupel der fachlichen Tabellen erfasst werden können. Das bedingte Vorkommen vollständiger Tupel stellt dagegen gerade die Grundlage von Variantenvielfalt dar, deren Beherrschung die Kernkompetenz der hier betrachteten Produktkonfiguratoren ist. Dementsprechend muss die zugehörige fachliche Modellierung so strukturiert sein, dass nach Abschnitt 2.2.2 zumindest

die Generierung kontextabhängiger Auswahlmöglichkeiten und die Ausprägung variabler Produktstrukturen unterstützt werden. Beispielsweise sind im Konfigurator von Porsche[1] die zur Auswahl stehenden Räder abhängig vom gewählten Modell verfügbar, wie dies in **Abbildung 4.1** für Boxster, 911 Turbo und 911 Carrera 4 skizziert ist. Dennoch werden typischerweise alle Radvarianten als Tupelausprägungen einer Tabelle persistiert, auf die dann jedes Fahrzeugmodell eine individuelle einschränkende Sicht bekommt.

		Modell		
		Boxster	911 Turbo	911 Carrera 4
Räder	18-Zoll Boxster Rad	x		
	19-Zoll Boxster S Rad	x		
	19-Zoll 911 Turbo II Rad		x	
	19-Zoll RS Spyder Rad		x	
	19-Zoll Carrera Rad			x
	20-Zoll Carrera S Rad	x		x
	20-Zoll Carrera Classic Rad	x		x
	20-Zoll Sport Design Rad			x
	20-Zoll Sport Techno Rad	x		x

Abbildung 4.1: Beispiel für gültige Kombinationen zwischen Modellen und Rädern

4.2 Aspektspezifische Daten

Wie durch Definition 3.1 charakterisiert, beeinflussen funktionale Aspekte die Daten eines Systems und führen zur Unterscheidung in fachliche nicht aspektabhängige und aspektspezifische Daten. Diese sind gemäß der Modularitäts-Anforderung des AOD-Paradigmas voneinander zu trennen, sowohl hinsichtlich der physischen Persistierungsebene als auch beim Umgang mit Aspekten auf logischer Ebene. Aus diesem Grund werden nachfolgend basierend auf [Pie11a, Pie11b] Begriffe und Konzepte eingeführt, die ein formales Gerüst für das hier betrachtete Referenzmodell und dessen in Kapitel 5 skizzierte prototypische Implementierung bieten. Mit jener Terminologie soll insbesondere die interne Verwaltung aspektspezifischer Attributwerte gemäß Klassifikation in Abschnitt 3.1.2 unterstützt werden. Zur besseren Veranschaulichung der Begrifflichkeiten kommt dabei das Anwendungsbeispiel aus Abschnitt 3.2 mit dem Relationenschema MODULE regelmäßig zum Einsatz.

4.2.1 Aspektschlüsselwertmengen und Aspektabhängigkeiten

Ergänzend zur eher visuellen Veranschaulichung funktionaler Aspekte im Zuge von Abschnitt 3.1.2.1 erfolgt nun mit Hilfe von Definition 4.1 und Definition 4.2 deren Formalisierung hinsichtlich der jeweiligen Aspektschlüssel sowie den Abhängigkeitsbeziehungen zu Attributen und Relationen.

[1] `http://www.porsche.com/germany/carconfiguratorgeneral`

▶ **Definition 4.1** *Ein funktionaler Aspekt A ist charakterisiert durch eine endliche Anzahl von diskreten Ausprägungen, der sogenannten* ***Aspektschlüsselwertmenge****:*

$$dom(A) = \{a_1, \ldots, a_m\}$$

Zudem existiere ein NULL-Symbol $\perp$, sodass für alle Aspekte $A_1, \ldots, A_n$ gilt:

$$\perp \notin \bigcup_{1 \leq i \leq n} dom(A_i)$$

Beispiel: Eine typische Aspektschlüsselwertmenge für den Aspekt „Sprache“ innerhalb des deutschsprachigen Raums sind die Locales $\{de\_DE, de\_AT, de\_CH\}$.

▶ **Definition 4.2** *Sei $\mathcal{A}$ die Menge aller Aspekte und $\mathcal{C}$ die Menge aller möglichen Attribute. Ein Attribut $C \in \mathcal{C}$ unterliegt einer* ***Aspektabhängigkeit*** *bezüglich eines Aspekts $A \in \mathcal{A}$, falls aspektspezifische Attributwerte von C durch A beeinflusst werden. Dabei seien die Aspektabhängigkeitsfunktionen $\delta : \mathcal{A} \times \mathcal{C} \to \{0, 1\}$ und $\Delta : \mathcal{C} \to \wp(\mathcal{A})$ wie folgt definiert:*

$$\delta(A, C) = \begin{cases} 1 : \textit{Attribut } C \textit{ hängt von Aspekt } A \textit{ ab} \\ 0 : \textit{sonst} \end{cases}$$
$$\Delta(C) = \{A : \delta(A, C) = 1\}$$

Sei weiterhin $R = (C_1, \ldots, C_n)$ ein Relationenschema. Dann ist Δ auch bezüglich der Relation R wie folgt definiert:

$$\Delta(R) = \bigcup_{1 \leq i \leq n} \Delta(C_i)$$

Bemerkung: Ein Attribut C gilt bereits dann abhängig von einem Aspekt A, d.h. $\delta(A, C)$ nimmt den Wert 1 an, sobald seine Attributwerte prinzipiell aspektspezifisch ausgeprägt sein *können*, selbst wenn in einem konkreten Datenbestand nur aspektunabhängige Attributwerte auftreten. Die Entscheidung über die Abhängigkeit wird letztlich durch die Anwendung bestimmt. Schließlich liefert die Funktion Δ ausgehend von einem Relationenschema die Menge all jener Aspekte, von denen die Attribute des Relationenschemas abhängig sind.

Beispiel: Die Anwendung der Abhängigkeitsfunktion Δ auf das Relationenschema MODULE und dessen Attribute nimmt entsprechend der Abbildung 3.5 folgende Werte an:

$$\begin{aligned} \Delta(\textsc{Key}) = \Delta(\textsc{RowID}) &= \varnothing \\ \Delta(\textsc{UserCreate}) = \Delta(\textsc{DateCreate}) &= \varnothing \\ \Delta(\textsc{Name}) = \Delta(\textsc{Description}) = \Delta(\textsc{Material}) &= \{\text{Sprache}, \text{Version}\} \\ \Delta(\textsc{Price}) &= \{\text{Region}, \text{Version}, \text{Staffel}\} \\ \Delta(\textsc{Norm}) &= \{\text{Region}, \text{Version}\} \\ \hline \Delta(\textsc{Module}) &= \{\text{Sprache}, \text{Region}, \text{Version}, \text{Staffel}\} \end{aligned}$$

4.2.2 Aspektsignaturen und signierte Tupel

Neben der prinzipiellen Spezifikation von Abhängigkeit zwischen Attributen und Aspekten gemäß Definition 4.2 (= Entwurfszeit), müssen für die Speicherung von aspektspezifischen Attributwerten (= Laufzeit) die tatsächlich vorhandenen Ausprägungen der relevanten Aspekte bekannt sein. Dazu wird in diesem Abschnitt der Begriff der Signatur eingeführt.

▶ **Definition 4.3** *Sei $R = (C_1, \ldots, C_n)$ ein Relationenschema und $\Delta(R)$ entsprechend Definition 4.2 die Menge jener Aspekte, von denen R abhängig ist. Sei weiterhin*

$$f_R : \Delta(R) \leftrightarrow \{1, \ldots, |\Delta(R)|\} \tag{4.1}$$

eine bijektive Abbildung. Dann wird

$$\Sigma(R, f_R) = \prod_{i=1}^{|\Delta(R)|} \Big(dom({f_R}^{-1}(i)) \cup \{\bot\} \Big) \tag{4.2}$$

*als die **Menge der Aspektsignaturen** über R mit Aspektnummerierung f_R bezeichnet. Ein Element $\sigma \in \Sigma(R, f_R)$ heißt dabei **Aspektsignatur**.*

Bemerkung:

(4.1) Die Definition der Funktion f_R hat einzig den Zweck der „Durchnummerierung“ aller Aspekte, die mit dem Relationenschema R gemäß $\Delta(R)$ in einer Abhängigkeitsbeziehung stehen. Aufgrund der Bijektivität von f_R ergibt sich die ein-eindeutige Zuordnung einer solchen Zahl für jeden dieser Aspekte.

(4.2) Ebenfalls durch die Eigenschaft der Bijektivität ergibt das Funktions-Urbild ${f_R}^{-1}(i)$ wieder eindeutig genau einen Aspekt. Somit wird also das kartesische Produkt über alle Ausprägungsmengen $dom({f_R}^{-1}(i)) = dom(A_i)$, jeweils erweitert um das NULL-Symbol $\bot$, derjenigen Aspekte A_i erzeugt, von den R abhängt.

Damit stellt $\Sigma(R, f_R)$ die Menge aller möglichen Aspektsignaturen für ein Relationenschema und dessen in Abhängigkeitsbeziehung stehenden Aspekte bezüglich einer Ordnung dar. Als abkürzende Schreibweise kann auch einfach $\Sigma(R)$ verwendet werden unter der Annahme, dass eine Ordnung *aller* Aspekte aus $\mathcal{A}$ vorliegt und diese nach Reduzierung irrelevanter Aspekte hinsichtlich R für Σ zum Einsatz kommt.

Beispiel: Ausgehend von der willkürlichen Aufzählung der Aspekte in Abschnitt 3.2.2 für das Relationenschema MODULE ergibt die Funktion f_{MODULE} folgende Ordnung:

$$f_{\text{MODULE}}(\text{Sprache}) = 1 \qquad f_{\text{MODULE}}(\text{Region}) = 2$$
$$f_{\text{MODULE}}(\text{Version}) = 3 \qquad f_{\text{MODULE}}(\text{Staffel}) = 4$$

Dadurch ist die Menge der Aspektsignaturen festgelegt auf

$$\begin{aligned}\Sigma(\text{MODULE}, f_{\text{MODULE}}) = {} & dom(\text{Sprache}) \cup \{\bot\} \times dom(\text{Region}) \cup \{\bot\} \\ & \times dom(\text{Version}) \cup \{\bot\} \times dom(\text{Staffel}) \cup \{\bot\}\end{aligned}$$

und das Tupel $(de, EU, 2, Standard) \in \Sigma(\text{MODULE}, f_{\text{MODULE}})$ stellt eine exemplarische Aspektsignatur dar, sofern deren Werte tatsächlichen Aspektausprägungen entsprechen.

▶ **Definition 4.4** *Sei $R = (C_1, \ldots, C_n)$ und $\ominus$ ein weiteres NULL-Symbol für das gilt:*

$$\ominus \notin \bigcup_{1 \leq i \leq n} dom(C_i)$$

Sei weiterhin R' ein gegenüber R um den einheitlichen NULL-Wert $\ominus$ erweitertes Relationenschema, $r(R')$ eine Relation und $t \in r(R')$ ein Tupel dieser Relation:

$$r(R') \subseteq dom(C_1) \cup \{\ominus\} \times \cdots \times dom(C_n) \cup \{\ominus\}$$
$$t = (v_1, \ldots, v_n)$$

Dann heißt $\sigma[v_1, \ldots, v_n]_R$ ***signiertes Tupel*** *bezüglich einer Aspektsignatur $\sigma \in \Sigma(R)$.*

Bemerkung: Das Konstrukt des signierten Tupels soll im weiteren Verlauf primär zur Repräsentation aspektspezifischer Daten als Ergänzung zum klassischen Tupel einer Relation verwendet werden. Deswegen beschränken sich nachfolgend signierte Tupel auf Relationenschemata mit ausnahmslos aspektabhängigen Attributen. Da typischerweise interne (Schlüssel-)Attribute wie MODULE.KEY oder MODULE.DATECREATE im Beispielmodell aus Abschnitt 3.2 nicht von Aspekten beeinflusst sind, muss die Identifizierung des betreffenden Tupels extern mitgegeben werden. Dies geschieht über den sogenannten *fachlichen Kontext*, welcher neben der fachlichen Basistabelle auch die RowID[2] desjenigen Tupels enthält, dessen aspektspezifische Ausprägung das signierte Tupel verkörpert.

Beispiel: Nachstehendes Tupel aus MODULE sei der fachliche Kontext:

('mod3141', 'Schraube', 0.37, 'Innensechskant', 'DIN 911',
'Edelstahl', 'Matthias Liebisch', 20120823, 3141)

Gemäß Abbildung 3.5 mit Festlegung der Aspektabhängigkeiten für MODULE lässt sich ein Relationenschema R definieren, welches nur aspektrelevante Attribute enthält:

$$R = (\text{NAME}, \text{PRICE}, \text{DESCRIPTION}, \text{NORM}, \text{MATERIAL})$$

Schließlich ist mit Angabe der gewünschten Aspektsignaturen

$$\sigma_1 = (en, UK, 18, Standard)$$
$$\sigma_2 = (fr, EU, 18, Standard)$$

die Erzeugung folgender signierter Tupel möglich, welche jedoch nur mit obigem fachlichen Kontext existieren können:

$$\sigma_1[\text{'screw'}, 0.33, \text{'hex socket'}, \text{'BS 4190'}, \text{'stainless steel'}]_R$$
$$\sigma_2[\text{'vis'}, 0.37, \text{'à six pans creux'}, \text{'ISO 4016'}, \text{'inox'}]_R$$

Zwecks Kompaktheit soll auch nachstehende Schreibweise erlaubt sein (hier beispielhaft für eine alternative Kombination aus Signatur und Relationenschema):

$$(en, 18)[\text{'screw'}, \text{'hex socket'}, \text{'low carbon steel'}]_{(\text{NAME,DESCRIPTION,MATERIAL})}$$

[2] Erläuterungen zum Konzept der RowID folgen in Abschnitt 4.3.2.

4.2.3 Aspektkontexte

Wie zuvor in Abschnitt 4.2.2 beschrieben, dienen Aspektsignaturen primär der Repräsentation aspektspezifischer Daten. Insbesondere sind diese immer bezüglich eines konkreten Relationenschemas definiert, was sich in Abschnitt 4.3 deutlich widerspiegelt. Neben der Speicherung und Verwaltung aspektabhängiger Daten soll das Referenzmodell aber auch den Zugriff auf diese geeignet unterstützen (siehe Abschnitt 4.4). Dazu wird nachfolgend der Begriff der Aspektkontexte eingeführt.

▶ **Definition 4.5** *Seien $dom(A_1), \ldots, dom(A_n)$ Aspektschlüsselwertmengen für n Aspekte. Die Menge c heißt* ***Aspektkontext*** *über $dom(A_1), \ldots, dom(A_n)$, falls gilt:*

$$c \subseteq \prod_{i=1}^{n} \Big(dom(A_i) \cup \{\perp\} \Big)$$

Ein Element $e \in c$ wird als ***Aspektkontextelement*** *bezeichnet. Enthält c nur genau ein Aspektkontextelement, dann heißt c* ***eindeutiger Aspektkontext****.*

Bemerkung: Ein Aspektkontextelement ist demnach ein n-Tupel, das allen n im System bekannten (und nicht nur den auf einer Relation wirkenden) Aspekten einen Aspektschlüsselwert (an der i-ten Stelle aus $dom(A_i)$) zuordnet oder den betreffenden Aspekt mittels NULL-Symbol $\perp$ unbelegt lässt. Daran anknüpfend wird ein Aspektkontext durch eine Menge beliebiger Aspektkontextelemente gebildet. Aspektkontexte haben damit die Aufgabe, für Zugriffs-Operationen die „Weltsicht“ – d.h. den Ausschnitt – auf die Daten aller Relationen des fachlichen Modells einzuschränken. Dadurch sind nur noch diejenigen aspektspezifischen Attributwerte relevant, welche mit einem gegebenen Kontext kompatibel sind. Dies trifft zu, wenn ein Kontextelement die Kombination von Aspektschlüsselwerten darstellt, unter der ein Attributwert zugeordnet wurde.

Beispiel: Die beiden Signaturen

$$\sigma_1 = (en, UK, 18, Standard)$$
$$\sigma_2 = (fr, EU, 18, Standard)$$

aus dem vorherigen Beispiel zu Definition 4.4 stellen auch zwei Aspektkontextelemente dar, weil jeder der vier Aspekte Sprache, Region, Version und Staffel mit jeweils einem gültigen Aspektschlüsselwert in beiden Tupeln vertreten ist. Ein möglicher resultierender Aspektkontext c ließe sich dann wie folgt definieren:

$$c = \{(en, UK, 18, Standard),\ (fr, EU, 18, Standard)\}$$

▶ **Definition 4.6** *Der über n Aspektschlüsselwertmengen eindeutige Aspektkontext*

$$c_{core} = \Big\{ (\underbrace{\perp, \ldots, \perp}_{n\ Elemente}) \Big\}$$

heißt ***Kernaspektkontext*** *und liefert durch die Belegung mit $\perp$ statt mit konkreten Aspektschlüsselwerten die Attributwerte der fachlichen Relationen ohne Aspekteinfluss.*

4.2.4 Aspektfilter

Die Verwendung der in Abschnitt 4.2.3 eingeführten Aspektkontexte verursacht durch deren explizite Mengen-Notation mit vollständig zu spezifizierenden Kontextelementen bezüglich aller registrierten Aspekte bereits zur Darstellung einfacher Sachverhalte einen beachtlichen Aufwand. Darüber hinaus besitzt die Semantik von Aspektkontexten keine Invarianz im Fall sich verändernder Aspektschlüsselwertmengen, was nachstehendes Beispiel verdeutlicht.

Beispiel: Für zwei Aspekte mit $dom(A_1) = \{a, b\}$ und $dom(A_2) = \{1, 2\}$ lässt sich durch den Aspektkontext

$$c = \{(a, \perp), (a, 1), (a, 2)\}$$

semantisch die Festlegung *„Schlüsselwert a im Aspekt A_1 bei gleichzeitiger Unabhängigkeit vom Aspekt A_2“* darstellen. Wird jedoch beispielsweise A_2 um den Schlüsselwert 3 ergänzt, existiert ein neues Element $(a, 3) \notin c$, um welches der Kontext c erweitert werden müsste, um dessen ursprüngliche Bedeutung hinsichtlich des Aspekts A_2 wieder herzustellen.

Deswegen wird in diesem Abschnitt eine eher deklarative Form präsentiert, die sich kompakter ohne Festlegung aller Kontextelemente und ihrer Komponenten sowie mit gewissen Freiheitsgraden spezifizieren lässt. Diese Reduzierung auf die eigentlichen Informationen soll gleichzeitig eine höhere Flexibilität und Robustheit gegenüber Veränderungen an Aspekten und deren Schlüsseln induzieren.

▶ **Definition 4.7** *Für die Aspekte eines Systems seien*

$$\bigwedge_{1 \leq i \leq n} dom(A_i) = \left\{a_1^i, \ldots, a_{m_i}^i\right\}$$

*die Aspektschlüsselwertmengen. Dann definiert sich für alle $1 \leq i \leq n$ und $1 \leq j \leq m_i$ die Menge der syntaktisch korrekten **Aspektfilter (AF)** induktiv wie folgt:*

$$keyValue\left(A_i, a_j^i\right) \in AF$$
$$keyNull\left(A_i\right) \in AF$$

Seien $f, f_1, \ldots, f_n \in AF$, so gilt weiterhin:

$$not\left(f\right) \in AF$$
$$and\left(f_1, \ldots, f_n\right) \in AF$$
$$or\left(f_1, \ldots, f_n\right) \in AF$$

Bemerkung: Basierend auf den elementaren Ausdrücken $keyValue$ und $keyNull$ sowie den zusammengesetzten Ausdrücken *and*, *or* und *not* ist eine syntaktische Definition von Aspektfiltern gegeben. Um diese als implizite Beschreibung von Aspektkontexten nutzen zu können, müssen beide Darstellung miteinander in Beziehung gesetzt werden. Die daraus resultierenden beiden Sichtweisen werden nachfolgend vorgestellt und beinhalten einerseits den aus einem Aspektfilter erzeugten Kontext als auch andererseits die Akzeptanz von Kontexten bezüglich gegebener Aspektfilter.

▶ **Definition 4.8** *Gegeben seien für die Aspekte* $A_1, \ldots, A_n$

$$A' = \prod_{i=1}^{n} \Big(dom(A_i) \cup \{\perp\} \Big) \tag{4.3}$$

sowie ein Aspektfilter-Ausdruck $f \in AF$*. Der durch* f ***induzierte Kontext*** $C_f \subseteq A'$ *bezüglich der* $A_1, \ldots, A_n$ *ist dann:*

$$C_f = \begin{cases} \{a \in A' : a_k = v\} & : \textit{falls } f = keyValue(A_k, v) \quad (4.4) \\ \{a \in A' : a_k = \perp\} & : \textit{falls } f = keyNull(A_k) \quad (4.5) \\ C_{f_1} \cap \cdots \cap C_{f_n} & : \textit{falls } f = and(f_1, \ldots, f_n) \quad (4.6) \\ C_{f_1} \cup \cdots \cup C_{f_n} & : \textit{falls } f = or(f_1, \ldots, f_n) \quad (4.7) \\ A' \setminus C_{f_1} & : \textit{falls } f = not(f_1) \quad (4.8) \end{cases}$$

Für einen Aspektfilter f *stellt der induzierte Kontext* C_f *also die Vereinigung aller Kontexte dar, die jener Filter akzeptiert. Andererseits wird ein Kontext* $c \subseteq A'$ *von einem Filter* $f \in AF$ ***akzeptiert*** *genau dann, wenn* C_f *und* c *mindestens ein Element gemeinsam haben. Damit lässt sich ein Filter* f *auch als charakteristische Funktion betrachten:*

$$\begin{aligned} f(c) \;&\in\; \{0, 1\} \\ f(c) = 1 \;&\Leftrightarrow\; c \cap C_f \neq \varnothing \end{aligned} \tag{4.9}$$

Bemerkung:

(4.3) Die Menge A' repräsentiert die Vereinigung aller Aspektkontexte bezüglich der A_i.

(4.4) Für den elementaren Aspektfilter $keyValue(A_k, v)$ mit $v \in A_k$ ist C_f die Menge aller möglichen Aspektkontextelemente mit Schlüsselwert v in der k-ten Komponente.

(4.5) Für den elementaren Aspektfilter $keyNull(A_k)$ ist C_f die Menge aller Aspektkontextelemente mit unbelegter k-ter Komponente ($\perp$-Symbol).

(4.6) Für den Aspektfilter-Ausdruck $and(f_1, \ldots, f_n)$ ist C_f die Schnittmenge aller induzierten Kontexte der einzelnen f_i-Operanden, d.h. die Menge jener Aspektkontextelemente, die in allen von den Operanden induzierten Kontexten existieren.

(4.7) Für den Aspektfilter-Ausdruck $or(f_1, \ldots, f_n)$ ist C_f analog zu (4.6) die Vereinigungsmenge aller induzierten Kontexte der einzelnen f_i-Operanden.

(4.8) Für den Aspektfilter-Ausdruck $not(f_1)$ ist C_f im Sinne einer Negation die inverse Menge des induzierten Kontexts von f_1 bezüglich der Gesamtmenge A'.

(4.9) Die Akzeptanz eines Kontexts c unter einem Aspektfilter f liegt vor, wenn die Schnittmenge zwischen c und dem durch f induzierten Kontext C_f mindestens ein gemeinsames Aspektkontextelement enthält.

Beispiel: Die Semantik des einleitend präsentierten Kontexts $c = \{(a, \perp), (a, 1), (a, 2)\}$ lässt sich nun mit dem Aspektfilter $keyValue(A_1, a)$ ausdrücken.

4.2.5 Uniforme, aufgefüllte, vereinbare und abgeleitete Tupel

Im letzten Teilabschnitt zur formalisierten Betrachtung aspektspezifischer Daten werden Mittel und Konstrukte definiert, die eine Vergleichbarkeit sowie weiterführende Prozessierung von aspektabhängigen Tupeln gleicher oder verschiedener Relationenschemata ermöglichen.

▶ **Definition 4.9** *Sei $R = (C_1, \ldots, C_n)$ ein Relationenschema und Δ die zugehörige Abhängigkeitsfunktion nach Definition 4.2. Wenn diesbezüglich die Eigenschaft*

$$\Delta(C_1) = \cdots = \Delta(C_n)$$

gilt, dann heißt R ***uniform****. Entsprechend wird in diesem Fall eine Relation $r(R)$* ***uniforme Relation*** *genannt und ein signiertes Tupel $t \in r(R)$ einer uniformen Relation als* ***uniformes Tupel*** *bezeichnet.*

Bemerkung: Beim Kriterium der Uniformität eines signierten Tupels ist zu beachten, dass die Entscheidung einzig und allein durch das zugehörige Relationenschema und dessen Aspektabhängigkeiten beeinflusst wird und nicht von den konkreten Attributwerten im Tupel. In einem uniformen Relationenschema sind also alle Attribute von den identischen Aspekten abhängig. Daraus folgt, dass die Signaturen aller Komponenten eines signierten Tupels untereinander sowie zur Signatur des Ausgangstupels gleich sind.

Beispiel: Gegeben sei das Relationenschema R mit Attributen aus dem Relationenschema MODULE und ein signiertes Tupel $t \in r(R)$:

$$\begin{aligned} R &= (\textsc{Name}, \textsc{Price}, \textsc{Description}, \textsc{Norm}, \textsc{Material}) \\ t &= (fr, EU, 18, \mathit{Standard})[\,\mathit{'vis'}, 0.37, \mathit{'à\ six\ pans\ creux'}, \mathit{'ISO\ 4016'}, \mathit{'inox'}]_R \end{aligned}$$

In diesem Szenario liegt keine Uniformität vor, weil R beispielweise mit NAME, PRICE und NORM Attribute enthält, die gemäß Abbildung 3.5 jeweils von verschiedenen Aspekten abhängig sind:

$$\begin{aligned} \Delta(\textsc{Name}) &= \{\text{Sprache}, \text{Version}\} \\ \Delta(\textsc{Price}) &= \{\text{Region}, \text{Version}, \text{Staffel}\} \\ \Delta(\textsc{Norm}) &= \{\text{Region}, \text{Version}\} \end{aligned}$$

Dagegen sind das Relationenschema $S \subseteq R$ und somit auch das signierte Tupel $t' \in r(S)$

$$\begin{aligned} S &= (\textsc{Name}, \textsc{Description}, \textsc{Material}) \\ t' &= (fr, 18)[\,\mathit{'vis'}, \mathit{'à\ six\ pans\ creux'}, \mathit{'acier\ doux'}]_S \end{aligned}$$

uniform aufgrund des folgenden Zusammenhangs:

$$\begin{aligned} \Delta(\textsc{Name}) &= \Delta(\textsc{Description}) = \Delta(\textsc{Material}) \\ &= \{\text{Sprache}, \text{Version}\} \end{aligned}$$

► **Definition 4.10** *Seien $R_1 = (C_1, \ldots, C_m)$ und $R_2 = (D_1, \ldots, D_n)$ Relationenschemata mit der Eigenschaft $m < n$. Sei weiterhin*

$$g : \{1, \ldots, m\} \to \{1, \ldots, n\}$$

eine streng monoton wachsende injektive Funktion, so dass gilt:

$$\bigwedge_{1 \le i \le m} C_i = D_{g(i)} \tag{4.10}$$

Darüber hinaus seien

$$\begin{aligned} k &= |\Delta(R_1)| \\ l &= |\Delta(R_2)| \\ f_{R_1} &: \Delta(R_1) \leftrightarrow \{1, \ldots, k\} \\ f_{R_2} &: \Delta(R_2) \leftrightarrow \{1, \ldots, l\} \\ (\alpha_1, \ldots, \alpha_k) &\in \Sigma(R_1, f_{R_1}) \\ (\beta_1, \ldots, \beta_l) &\in \Sigma(R_2, f_{R_2}) \end{aligned} \tag{4.11}$$

Schließlich seien t_1 und t_2 zwei signierte Tupel

$$\begin{aligned} t_1 &= (\alpha_1, \ldots, \alpha_k)[c_1, \ldots, c_m]_{R_1} \\ t_2 &= (\beta_1, \ldots, \beta_l)[d_1, \ldots, d_n]_{R_2} \end{aligned}$$

für die gilt:

$$\bigwedge_{1 \le j \le n} d_j = \begin{cases} c_i & \textit{falls} \bigvee\limits_{1 \le i \le m} g(i) = j \\ \ominus & \textit{sonst} \end{cases} \tag{4.12}$$

$$\bigwedge_{1 \le j \le l} \beta_j = \begin{cases} \alpha_i & \textit{falls} \bigvee\limits_{1 \le i \le k} {f_{R_1}}^{-1}(i) = {f_{R_2}}^{-1}(j) \\ \perp & \textit{sonst} \end{cases} \tag{4.13}$$

Dann wird t_2 als ein von t_1 entstammendes auf R_2 ***aufgefülltes Tupel*** *bezeichnet.*

Bemerkung:

(4.10) Diese Voraussetzung ist erfüllt, wenn alle Attribute von R_1 auch in R_2 existieren und zudem in beiden Relationenschemata mit derselben Reihenfolge[3] auftreten.

(4.11) Festlegung zweier Signaturen $(\alpha_1, \ldots, \alpha_k)$ und $(\beta_1, \ldots, \beta_l)$ passend zu R_1 und R_2.

(4.12) Hiermit werden die Attributwerte von Tupel t_2 bezüglich Tupel t_1 so festgelegt, dass an Stellen mit gleichen Attributen in beiden Tupeln auch gleiche Werte stehen und in allen anderen Fällen das NULL-Symbol $\ominus$.

[3] Im Sinne einer gleichen, z.B. lexikografischen, Ordnung bei der Schemadefinition.

(4.13) Analog zu den Attributwerten findet hier die Festlegung der Tupelsignatur von t_2 bezüglich der von t_1 so statt, dass an Stellen mit gleichen Aspekten in beiden Signaturen auch gleiche Werte stehen und sonst für Aspekte, von denen nur t_2 abhängt, diese Abhängigkeit mittels NULL-Symbol $\perp$ leer bleibt.

Zusammenfassend kann das Auffüllen von Tupeln als eine Methode betrachtet werden, um die Kompatibilität zwischen einem Tupel und einem Relationenschema herzustellen. Voraussetzung hierfür ist, dass alle Attribute des Tupels auch im Relationenschema vorkommen. Darüber hinaus existierende Attribute führen zur Auffüllung der resultierenden „Lücken“ bei den Attributwerten des Tupels und bei den Aspektbelegungen der zugeordneten Tupelsignatur mit den entsprechenden Nullwerten $\ominus$ beziehungsweise $\perp$.

Beispiel: Gegeben seien die beiden Relationenschemata R und S mit Attributen aus dem Relationenschema MODULE:

$$R = (\text{NAME}, \text{PRICE}, \text{DESCRIPTION}, \text{NORM}, \text{MATERIAL})$$
$$S = (\text{NAME}, \text{DESCRIPTION}, \text{MATERIAL})$$

Dann lässt sich das signierte Tupel $t_1 \in r(S)$

$$t_1 = (fr, 18)[\textit{'vis'}, \textit{'à six pans creux'}, \textit{'inox'}]_S$$

als ein bezüglich R aufgefülltes Tupel $t_2 \in r(R)$ wie folgt darstellen:

$$t_2 = (fr, \perp, 18, \perp)[\textit{'vis'}, \ominus, \textit{'à six pans creux'}, \ominus, \textit{'inox'}]_R$$

▶ **Definition 4.11** *Seien $R_1 = (C_1, \ldots, C_m)$ und $R_2 = (D_1, \ldots, D_n)$ Relationenschemata und $(\alpha_1, \ldots, \alpha_k)[c_1, \ldots, c_m]_{R_1}$ sowie $(\beta_1, \ldots, \beta_l)[d_1, \ldots, d_n]_{R_2}$ jeweils passende signierte Tupel. Die beiden Tupel heißen **vereinbar**, wenn gilt:*

$$\bigwedge_{1\le i\le k} \bigwedge_{1\le j\le l} \quad {f_{R_1}}^{-1}(i) = {f_{R_2}}^{-1}(j) \longrightarrow \alpha_i = \beta_j \tag{4.14}$$

$$\bigwedge_{1\le i\le m} \bigwedge_{1\le j\le n} \quad C_i = D_j \longrightarrow c_i = d_j \tag{4.15}$$

$$\{(C_1, c_1), \ldots, (C_m, c_m)\} \neq \{(D_1, d_1), \ldots, (D_n, d_n)\} \tag{4.16}$$

Bemerkung:

(4.14) Sind die beiden Tupel vom selben Aspekt abhängig (dargestellt durch das Urbild der Funktion f_R gemäß Definition 4.3), dann müssen die betreffenden Positionen in beiden Signaturen mit einem identischen Aspektschlüsselwert belegt sein.

(4.15) Besitzen die Relationenschemata gemeinsame Attribute, dann müssen die Tupel an den entsprechenden Positionen die gleichen Attributwerte aufweisen.

(4.16) Diese Bedingung schließt semantisch identische Tupel als unvereinbar aus, d.h. auch Tupel die sich nur aufgrund der Reihenfolge ihrer gleichen Attribute unterscheiden.

Das Kriterium der Vereinbarkeit liegt also vor, wenn zwei Tupel einerseits nicht semantisch gleich sind und sich andererseits hinsichtlich ihrer Attributwerte bzw. Aspektsignaturen nicht widersprechen.

Beispiel: Gegeben seien die beiden Relationenschemata R und S mit Attributen aus dem Relationenschema MODULE:

$$R = (\text{NAME}, \text{PRICE})$$
$$S = (\text{NAME}, \text{MATERIAL})$$

Dann sind die signierten Tupel $t_1 \in r(R)$ und $t_2 \in r(S)$

$$t_1 = (fr, EU, 18, Standard)[\textit{'vis'}, 0.37]_R$$
$$t_2 = (fr, 18)[\textit{'vis'}, \textit{'inox'}]_S$$

miteinander vereinbar, da sowohl das gemeinsame Attribut NAME den gleichen Attributwert besitzt als auch die übereinstimmenden Aspekte Sprache und Version in der Signatur die jeweils gleiche Ausprägung zeigen. Unterschiedliche Aspekte und Attribute sind für die Vereinbarkeit irrelevant, deswegen sind trivialerweise zwei signierte Tupel ohne gemeinsamen Aspekt immer vereinbar. Dagegen ist $t_3 \in r(S)$

$$t_3 = (en, 18)[\textit{'screw'}, \textit{'stainless steel'}]_S$$

weder mit t_1 noch mit t_2 vereinbar.

▶ **Definition 4.12** *Seien $R_1 = (C_1, \ldots, C_m)$ und $R_2 = (D_1, \ldots, D_n)$ Relationenschemata und $t_1 = (\alpha_1, \ldots, \alpha_k)[c_1, \ldots, c_m]_{R_1}$ sowie $t_2 = (\beta_1, \ldots, \beta_l)[d_1, \ldots, d_n]_{R_2}$ zugehörige vereinbare signierte Tupel.*
Das Tupel $t' = (\gamma_1, \ldots, \gamma_p)[e_1, \ldots, e_q]_{(E_1,\ldots,E_q)}$ heißt aus t_1 und t_2 ***abgeleitet*** *bzw.*

$$(t_1, t_2) \vdash t'$$

falls gilt:

$$\{C_1, \ldots, C_m\} \cup \{D_1, \ldots, D_n\} = \{E_1, \ldots, E_q\} \tag{4.17}$$

$$\bigwedge_{1\le i\le q} \bigwedge_{1\le j\le m} E_i = C_j \longrightarrow e_i = c_j \tag{4.18}$$

$$\bigwedge_{1\le i\le q} \bigwedge_{1\le j\le n} E_i = D_j \longrightarrow e_i = d_j \tag{4.19}$$

$$\bigwedge_{1\le i\le p} \bigwedge_{1\le j\le k} {f_{(E_1,\ldots,E_q)}}^{-1}(i) = {f_{R_1}}^{-1}(j) \longrightarrow \gamma_i = \alpha_j \tag{4.20}$$

$$\bigwedge_{1\le i\le p} \bigwedge_{1\le j\le l} {f_{(E_1,\ldots,E_q)}}^{-1}(i) = {f_{R_2}}^{-1}(j) \longrightarrow \gamma_i = \beta_j \tag{4.21}$$

Bemerkung:

(4.17) Die Vereinigung der Attribute aus den Relationenschemata R_1 und R_2 bildet die Attributmenge für das Relationenschema des abgeleiteten Tupels t'.

(4.18) & (4.19) Der Wert eines Attributs im Ergebnistupel muss dem Wert des zugehörigen Attributs im Basistupel entsprechen. Sollte das Attribut sowohl in t_1 als auch t_2 vorkommen, hat es dort aufgrund der Vereinbarkeit beider Tupel den gleichen Wert.

(4.20) & (4.21) Die Belegung eines Aspekts in der Signatur von t' ergibt sich aus dem Aspektschlüsselwert desselben Aspekts in der Signatur des Basistupels. Sollten sowohl t_1 als auch t_2 von diesem Aspekt abhängig sein, ist dieser in beiden Signaturen aufgrund der Vereinbarkeit beider Tupel mit dem gleichen Wert belegt.

Die Informationen zweier vereinbarer signierter Tupel sind nach einem solchen Ableitungsvorgang also vollständig im erzeugten, ebenfalls signierten, Tupel gespeichert. Ebenso lässt sich dieser Vorgang beispielsweise zur Erzeugung uniformer Tupel umkehren, was bei der Transformation zwischen Zugriffsmodell und Persistenzmodell in Abschnitt 4.4.1 eine wichtige Rolle spielen wird.

Beispiel: Gegeben seien die beiden signierten und vereinbaren Tupel aus dem Beispiel von Definition 4.11:

$$t_1 = (\mathit{fr}, \mathit{EU}, 18, \mathit{Standard})[\,\mathit{'vis'}, 0.37]_{(\textsc{Name},\textsc{Price})}$$
$$t_2 = (\mathit{fr}, 18)[\,\mathit{'vis'}, \mathit{'inox'}]_{(\textsc{Name},\textsc{Material})}$$

Dann können t_1 und t_2 wie folgt abgeleitet werden:

$$(t_1, t_2) \vdash (\mathit{fr}, \mathit{EU}, 18, \mathit{Standard})[\,\mathit{'vis'}, 0.37, \mathit{'inox'}]_{(\textsc{Name},\textsc{Price},\textsc{Material})}$$

4.2.6 Fazit

Die vorherigen Abschnitte haben gezeigt, dass mit Aspektsignaturen gemäß Definition 4.3 ein adäquates und formales Konstrukt zur Charakterisierung aspektspezifischer Daten existiert. Zur Gewährleistung der Lokalitäts-Anforderung des AOD-Paradigmas ist eine effiziente und generische Speicherung jener Signaturen auf Basis uniformer Relationen (siehe Abschnitt 4.2.5) erforderlich. Allerdings hängt die Uniformität von den jeweiligen Aspektzuordnungen zu den Attributen eines Relationenschemas ab und unterliegt damit der Schema-Evolution sowie dem Prozess der An- und Abkopplung funktionaler Aspekte. Größtmögliche Flexibilität bezüglich dieser Dynamik bietet deswegen die Verwaltung von Signaturen für individuelle Attribute als die trivialsten uniformen Tupel. Dennoch sind insbesondere durch Konzepte wie aufgefüllte und abgeleitete Tupel Signaturen für beliebige Relationenschemata darstellbar. Die hierfür notwendigen Datenstrukturen werden im nachfolgenden Abschnitt vorgestellt.

4.3 Persistenzmodell

Ein zentraler Bestandteil des hier beschriebenen Persistenzmodells stellt das bereits in Abschnitt 3.4.1 erwähnte EAV-Konzept dar, dessen Ursprünge, Funktionsprinzipien und Bezüge zum AOD-Paradigma nachfolgend in Abschnitt 4.3.1 im Detail vorgestellt werden. Anschließend findet eine Beschreibung der relationalen Strukturen durch Abschnitt 4.3.2 statt, deren Zusammenspiel in Abschnitt 4.3.3 erläutert wird.

4.3.1 EAV-Konzept

Zuordnungen von Werten zu Attributen werden im klassischen Relationenschema durch Spalten einer Tabelle repräsentiert, die Gesamtheit aller Attributwerte einer Tabellenzeile bildet gemäß Abbildung 3.6 einen Datensatz. Dagegen äußert sich jedes Attribut-Wert-Paar zu einem Objekt bzw. einer Entität im EAV-Modell (Entity-Attribute-Value) als eine eigene Tabellenzeile, der ursprüngliche Datensatz verteilt sich damit über mehrere solcher Zeilen. Wie bereits in Abschnitt 3.4.1 kurz erläutert, stellt für diesen Modellierungsansatz die EAV-Tabelle mit folgendem Aufbau das zentrale Element dar [NMC+99].

- *entity*: Bezeichnung bzw. Identifikation eines Objekts
- *attribute*: Bezeichnung bzw. Identifikation eines Attributs, typischerweise als Fremdschlüssel in eine andere Relation zur Beschreibung verfügbarer Attribute
- *value*: Wert, welcher dem Objekt zum angegebenen Attribut zugewiesen ist

Die Grundlage dieser Speicherform findet sich im Konzept der assoziativen Arrays zur Verwaltung von Attribut-Wert-Paaren, welche bereits 1958 mit Lisp eingeführt wurden und heutzutage in allen gängigen Programmiersprachen verfügbar sind, wie beispielweise als Klasse `java.util.HashMap` in Java. Darüber hinaus basieren sogenannte *Key-Value-Stores* innerhalb der seit einigen Jahren populären Datenbank-Bewegung Not only SQL (NoSQL) auf den gleichen Prinzipien, einen Überblick hierzu bieten [Cat10, Ste12].

4.3.1.1 Beispiel

Die Speicherung klinischer Untersuchungs-Daten im Sinne einer digitalen Patientenakte stellt ein typisches Anwendungsgebiet des EAV-Konzepts dar [FHJ+90, LKH11]. Einerseits existieren hunderte von potentiell körperlichen und psychischen Kriterien, von denen ein Arzt aber nur diejenigen gegenüber einem Patienten abfragen wird, welche aus seiner Sicht im vorliegenden Fall relevant sind. Andererseits verändert sich die Menge der Kriterien permanent, beispielsweise durch neue Messtechniken oder Forschungsergebnisse.

Nach klassischem relationalen Modellierungs-Schema entsteht üblicherweise eine Tabelle mit der Patienten-ID, einem Untersuchungs-Datum sowie allen Kriterien als Spalten. Dadurch wäre die Tabelle mit sehr vielen NULL-Werten gefüllt (entspricht den nicht abgefragten Kriterien) und jedes neue Kriterium würde eine Anpassung dieses höchst inflexiblen Datenbank-Schemas und aller davon abhängigen Anwendungen erfordern.

Patient

PatID	Vorname	Nachname	Geburtstag
0815	Max	Mustermann	01.01.1970
3141	Erna	Musterfrau	12.12.1960

Kriterium

KID	Beschreibung
1	Größe (in cm)
2	Gewicht (in kg)
3	Temperatur (in C)
4	Puls
5	Diabetes

Befund

PatID	Datum	KID	Wert
0815	01.04.2012	3	38
0815	01.04.2012	4	120
0815	01.04.2012	5	nein
3141	17.12.1999	2	150
3141	17.12.1999	5	ja

Abbildung 4.2: Grundkonzept des EAV-Modells am Beispiel der digitalen Patientenakte

Durch Nutzung des EAV-Konzepts wie in **Abbildung 4.2** beispielhaft skizziert können beide genannten Probleme vermieden werden. Dabei stellt Befund die zentrale EAV-Relation dar, in der zu einem Patienten (PatID), einer Untersuchung (Datum) und einem

Kriterium (KID) ein konkreter Wert vermerkt ist. Somit werden nur tatsächlich abgefragte Kriterien gespeichert und eine Erweiterung dieser ist durch Pflege neuer Datensätze in der Tabelle KRITERIUM völlig ohne Schemaänderungen möglich. Allerdings bringt die Verwendung des EAV-Konzepts neue Herausforderungen und unerwünschte Konsequenzen mit, welche nachfolgend in Abschnitt 4.3.1.2 betrachtet werden.

4.3.1.2 Bewertung

Das EAV-Konzept bietet folgende Vorteile bzw. könnte bei diesen Anforderungen zur Anwendung kommen:

Speichereffizienz: Da in der EAV-Tabelle nur tatsächliche, d.h. von NULL verschiedene, Attributwerte verwaltet werden, eignet sich dieses Konzept zur optimierten Speicherung sogenannter dünn besetzter Tabellen.

Flexibilität: Müssen neue Attribute zu den gespeicherten Entitäten erfasst werden, ist dies im Gegensatz zur klassischen Schema-Änderung ohne strukturelle Anpassungen möglich. Stattdessen können zusätzliche Attribute einfach als Datenzeilen erfasst werden.

Semistrukturiertheit: Aus den bisherigen Eigenschaften folgt für das EAV-Modell automatisch die Unterstützung semistrukturierter Daten wie im Fall von XML (siehe Abschnitt 3.4.3), d.h. es können abweichende Attribut-Mengen für verschiedene Entitäts-Ausprägungen verwaltet werden.

Unbeschränktheit: Aufgrund der Abbildung von Tabellenspalten als Zeilen können typische Obergrenzen für die Anzahl von Attributen je Tabelle in den DBMS-Produkten umgangen werden. Beispielsweise sind in IBM DB2 maximal 1012 Spalten erlaubt [IBM12] während Oracle 11g höchstens 1000 Spalten unterstützt [Ora12].

Mit Gewährleistung der aufgeführten positiven Eigenschaften sind allerdings auch einige Einschränkungen und Problematiken verbunden.

Integritätsgefährdung: Einige der in RDBMS etablierten Konzepte zur Integritätssicherung greifen nicht mehr bzw. sind nicht mehr sinnvoll anwendbar. Hierzu zählen insbesondere spaltenorientierte Mechanismen, wie beispielsweise eine Fremdschlüssel-Definition auf der Value-Spalte einer EAV-Tabelle.

Schwache Typisierung: Durch die Konsolidierung der unterschiedlichsten Attribute innerhalb der EAV-Tabelle muss die Value-Spalte zwangsläufig Werte mit den verschiedensten Datentypen aufnehmen können. Dies lässt sich entweder durch einen möglichst generischen Datentyp (beispielsweise VARCHAR) inklusive zusätzlicher Meta-Daten zu jedem Attribut realisieren oder durch Definition eigenständiger EAV-Tabellen für jeden relevanten Datentyp lösen [NMC+99, Sch11]. In jedem Fall gestaltet sich die Anfrage-Generierung aufgrund der Datenverteilung aufwändiger.

Anfrageverarbeitung: Abhängig von der Art einer Anfrage und ihrer erforderlichen Ergebnis-Aufbereitung kann deren Verarbeitung die intensive Nutzung des JOIN-Operators nach sich ziehen. Dieser führt bei entsprechenden Mengengerüsten zu signifikanten Verzögerungen in der Anfrage-Performance.

4.3.1.3 Anwendung im Referenzmodell

Der Einsatz des EAV-Konzepts ist im Referenzmodell für die AOD auf genau eine Relation AspectValue beschränkt, welche anschließend in Abschnitt 4.3.2 im Detail vorgestellt wird. Hiermit sind bereits die zwei wichtigsten in Abschnitt 3.3 formulierten Anforderungen des AOD-Paradigmas gewährleistet. Einerseits können in dieser EAV-Relation aspektspezifische Daten ohne Beeinflussung der fachlichen Relationen gespeichert werden (Modularität), stattdessen lässt sich eine Entität aus einer anderen Relation wie zuvor beschrieben referenzieren. Andererseits ist gemäß der Flexibilitäts-Eigenschaft die Definition von Aspekt-Abhängigkeiten ohne Schema-Änderungen möglich (Lokalität). Die mit dem EAV-Konzept verbundenen zuvor genannten negativen Auswirkungen auf die Anfrageverarbeitung sowie Benutzerfreundlichkeit werden separat in Abschnitt 4.4 betrachtet.

4.3.2 Aufbau und Strukturen

Dieser Abschnitt beschreibt die konkreten Strukturen und deren Zusammenhänge im relationalen Referenzmodell für das AOD-Paradigma basierend auf [Lie10b, Pie11a]. Der logische Modellentwurf ohne Datentypen ist in **Abbildung 4.3** dargestellt. Die darin definierten Relationen dienen entweder der Verwaltung von Metadaten zu Aspekten und fachlichen Relationen (*Aspect Catalog*) oder speichern die konkreten aspektspezifischen Daten (*Aspect Weaving*). Nachfolgend werden die einzelnen Tabellen genauer erläutert.

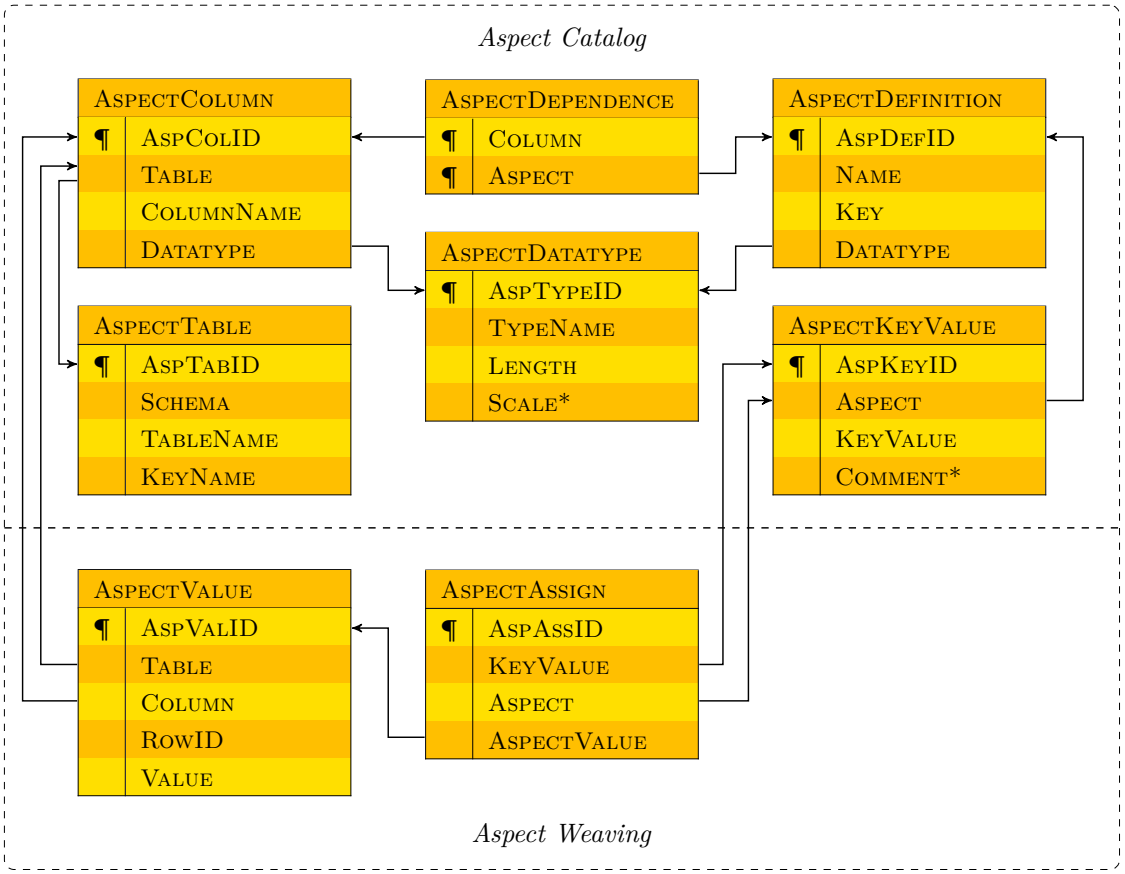


Abbildung 4.3: Logischer Entwurf des relationalen Referenzmodells

4.3.2.1 AspectTable

Die Tabelle AspectTable verwaltet Informationen über relevante Tabellen eines existierenden fachlichen Datenmodells als Grundlage für die Definition von Aspektabhängigkeiten. Zu einem generierten Primärschlüssel (AspTabID) werden das Schema (Schema) und der Name (TableName) zur Identifizierung einer Tabelle innerhalb der Datenbank angegeben. Die Nutzung der in DBMS-Produkten vorhandenen Katalogdaten ist jedoch aufgrund unterschiedlicher Implementierungen nicht möglich.

Zusätzlich speichert KeyName den Namen desjenigen ein-elementigen ganzzahligen Schlüssel-Attributs der fachlichen Tabelle, welches zur späteren Referenzierung eines Datensatzes bei der Verwaltung aspektspezifischer Daten (siehe Abschnitt 4.3.2.7) verwendet werden soll. Existiert ein solches Attribut in der Fachtabelle noch nicht, muss es zusätzlich angelegt werden – die einzige notwendige Anpassung, welche das Referenzmodell neben den in Abbildung 4.3 dargestellten Relationen mit sich bringt. Dabei ist dieses über AspectValue.RowID referenzierte Attribut nicht mit dem RowID-Konzept zu verwechseln, welches in einigen RDBMS zur Adressierung von Datensätzen verwendet wird (siehe „ROWID Pseudocolumn“ bei Oracle [Ora12] oder auch das Konzept der Globally Unique Identifier (GUID) bei MS SQL Server im Kontext der Replikation).

4.3.2.2 AspectColumn

Die Tabelle AspectColumn verwaltet Informationen über Attribute von fachlichen Tabellen, auf deren Basis konkrete Aspektabhängigkeiten definiert sowie aspektspezifische Daten zu fachlichen Attributwerten zugeordnet werden können. Zu einem generierten Primärschlüssel (AspColID) sind die Bezeichnung des Attributs (ColumnName), die zugehörige Tabelle per Fremdschlüssel (Table) und der ebenfalls per Fremdschlüssel verknüpfte Datentyp (DataType) hinterlegt (siehe Abschnitt 4.3.2.3).

4.3.2.3 AspectDatatype

Die Tabelle AspectDatatype verwaltet Informationen über Datentypen zu Attributen aus fachlichen Tabellen sowie den Schlüsselwerten funktionaler Aspekte. Zu einem generierten Primärschlüssel (AspTypeID) werden die Bezeichnung des Datentyps (TypeName), die Anzahl verfügbarer Stellen bzw. Zeichen (Length) und seine Genauigkeit im Fall von Fließkomma-Zahlen (Scale) gespeichert. Damit können grundlegende SQL-Datentypen wie VARCHAR, INT oder DECIMAL abgebildet und wie in Abschnitt 4.3.1.2 vorgeschlagen zur korrekten Auswertung des Attributs AspectValue.Value verwendet werden.

4.3.2.4 AspectDefinition

Die Tabelle AspectDefinition verwaltet Informationen zu funktionalen Aspekten, welche als querschnittliche Belange im Sinne der Begriffsprägung in Abschnitt 3.1.1 für ein Anwendungssystem relevant sind. Zu einem generierten Primärschlüssel (AspDefID) werden der Name des Aspekts (Name), der zugehörige Aspektschlüssel (Key) und dessen Datentyp über einen Fremdschlüssel (Datatype) angegeben. Somit lassen sich beispielsweise zweibuchstabige Locales als Schlüssel des funktionalen Aspekts „Mehrsprachigkeit“ definieren.

4.3.2.5 AspectKeyValue

Die Tabelle AspectKeyValue verwaltet Informationen zu Ausprägungen von Aspektschlüsseln und beinhaltet damit die Aspektschlüsselwertmengen aus Definition 4.1. Zu einem generierten Primärschlüssel (AspKeyID) werden der Schlüsselwert (KeyValue), ein optionaler Kommentar (Comment) sowie über einen Fremdschlüssel der zugehörige Aspekt (Aspect) gespeichert.

Zur Unterstützung numerischer und alphanumerischer Aspektschlüsselwerte hat das Attribut KeyValue den Datentyp VARCHAR, der tatsächliche Datentyp des funktionalen Aspekts ist bei dessen Definition festzulegen (siehe Abschnitt 4.3.2.4) und kann anschließend beim Zugriff auf Aspektschlüsselwerte ausgewertet werden.

4.3.2.6 AspectDependence

Die Tabelle AspectDependence verwaltet die explizit hinterlegten Abhängigkeiten zwischen Attributen des fachlichen Modells und funktionalen Aspekten gemäß Definition 4.2. Dementsprechend enthält ein Tupel über zugehörige Fremdschlüssel sowohl den betreffenden Aspekt (Aspect) als auch das davon abhängige Attribut (Column). Beide zusammen bilden den Primärschlüssel der Relation.

4.3.2.7 AspectValue

Die Tabelle AspectValue verwaltet alle aspektspezifischen Werte für signierte Tupel gemäß Definition 4.4. Zu einem generierten Primärschlüssel (AspValID) wird der aspektabhängige Wert (Value) bezüglich eines Attributwerts (Fremdschlüssel Column sowie RowID) einer Tabelle des fachlichen Modells (Fremdschlüssel Table) gespeichert. Die eigentliche Zuordnung des Aspektschlüsselwerts erfolgt davon unabhängig über eine separate Tabelle (siehe Abschnitt 4.3.2.8).

Aufgrund des verwendeten EAV-Konzepts bei der betrachteten Tabelle benötigt das Attribut Value den Datentyp VARCHAR zur Aufnahme von numerischen und alphanumerischen aspektspezifischen Werten. Der tatsächliche Datentyp kann über die Metadaten der zugehörigen Tabellenspalte ermittelt und ausgewertet werden. Weiterhin gilt, dass das Attribut RowID wie in Abschnitt 4.3.2.1 diskutiert nur bezüglich der referenzierten fachlichen Tabelle Schlüsseleigenschaft besitzt, nicht aber in Tabelle AspectValue.

4.3.2.8 AspectAssign

Die Tabelle AspectAssign verwaltet Zuordnungen aspektspezifischer Werte (siehe Abschnitt 4.3.2.7) zu Aspektschlüsselwerten und realisiert damit effektiv die Aspektsignaturen aus Definition 4.3. Zu einem generierten Primärschlüssel (AspAssID) werden jeweils über Fremdschlüssel der aspektspezifische Wert (AspectValue), ein Aspektschlüsselwert (KeyValue) sowie der Aspekt selbst (Aspect) hinterlegt. Dadurch erst ist die Möglichkeit gegeben, einen aspektabhängigen Wert auch unter Einfluss von **beliebig vielen** funktionalen Aspekten zu definieren. Anhand konkreter Beispieldaten wird in Abschnitt 4.3.3 dieser Zusammenhang veranschaulicht.

4.3.2.9 Prämissen und Integritätsbedingungen

Wie in Abschnitt 4.3.2.1 erläutert, setzt das Referenzmodell ein ganzzahliges einelementiges Schlüsselattribut in allen Tabellen des fachlichen Modells voraus, die von funktionalen Aspekten beeinflusst werden sollen. Darüber hinaus sind neben den zuvor beschriebenen und in Abbildung 4.3 dargestellten Fremdschlüssel-Beziehungen nachfolgend aufgeführte Integritätsbedingungen für eine konsistente Persistierung aspektspezifischer Daten zu gewährleisten. Die zur Formulierung von Ausdrücken verwendete Relationenalgebra orientiert sich dabei an [Cod70].

Bedingung C1 ist erfüllt, wenn gilt:

(TABLE, COLUMN, ROWID, VALUE) ist in Tabelle ASPECTVALUE eindeutig.

Dadurch wird die Erfassung von Duplikaten als aspektspezifische Daten zu einem konkreten Attributwert einer fachlichen Tabelle vermieden. Für die einfachere Referenzierung und Verarbeitung im Referenzmodell ist jedoch diesen als Schlüsselkandidat definierten Attributen der künstliche Primärschlüssel ASPVALID zugeordnet.

Bedingung C2 ist erfüllt, wenn gilt:

(ASPECT, ASPECTVALUE) ist in Tabelle ASPECTASSIGN eindeutig.

Diese Integritätsbedingung gewährleistet, dass ein aspektspezifischer Wert jedem funktionalen Aspekt nur genau einmal zugeordnet werden kann, unabhängig vom konkreten Aspektschlüsselwert. Notwendig ist diese Einschränkung für eine saubere Auswertungs-Semantik bei Interaktion mehrerer Aspekte (siehe Abschnitt 4.4). Dadurch ist es beispielsweise nicht möglich, den Ident eines deutschen Übersetzungstexts auch für andere Locales (AT, CH) zu nutzen. Stattdessen muss der betreffende Text jeweils als neuer aspektspezifischer Wert in ASPECTVALUE erfasst werden.

Bedingung C3 ist erfüllt, wenn gilt:

$$\underset{Column,Aspect}{\pi}\left(AspectAssign \underset{AspectValue=AspValID}{\bowtie} AspectValue\right) \subseteq \underset{Column,Aspect}{\pi}(AspectDependence)$$

Hierdurch wird sichergestellt, dass in den Tabellen ASPECTASSIGN und ASPECTVALUE nur Daten existieren, die bezüglich der projizierten Tupel (COLUMN, ASPECT) einen entsprechenden Eintrag in der Tabelle ASPECTDEPENDENCE besitzen. Anders ausgedrückt: aspektspezifische Daten lassen sich nur gemäß der zuvor definierten Aspekt-Abhängigkeiten hinterlegen.

Bedingung C4 ist erfüllt, wenn gilt:

$$\underset{AspValID}{\pi}(AspectValue) \subseteq \underset{AspectValue}{\pi}(AspectAssign)$$

Mit dieser Integritätsbedingung ist garantiert, dass jeder Datensatz in der Tabelle ASPECTVALUE mindestens von einem Datensatz in der Tabelle ASPECTASSIGN über den Fremdschlüssel bezüglich ASPVALID referenziert wird. Dadurch lassen sich in der Tabelle ASPECTVALUE nur aspektspezifische Attributwerte erfassen bzw. als solche auswerten.

Bedingung C5 ist erfüllt, wenn gilt:

$$\bigwedge_{t,u \in AspectValue} \Big[(t_{RowID} = u_{RowID} \wedge t_{Column} = u_{Column} \wedge t_{AspValID} \neq u_{AspValID}) \longrightarrow \Big(\underset{Aspect,KeyValue}{\pi} \Big(\underset{AspValID=t_{AspValID}}{\sigma} AspectAssign \Big) \neq \underset{Aspect,KeyValue}{\pi} \Big(\underset{AspValID=u_{AspValID}}{\sigma} AspectAssign \Big) \Big) \Big]$$

Diese Integritätsbedingung stellt sicher, dass zwei verschiedene Tupel in der Tabelle AspectValue mit Referenzierung des identischen Attributwerts einer Fachtabelle auch unterschiedliche Zuordnungen von Aspektschlüsselwerten in der Tabelle AspectAssign besitzen. Dadurch wird gewährleistet, dass es zu einem Attributwert einer fachlichen Tabelle und einer konkreten Aspektschlüsselwert-Kombination höchstens einen aspektabhängigen Wert geben kann.

4.3.3 Wirkung und Funktionsweise

Anhand des Anwendungsbeispiels aus Abschnitt 3.2 und der aspektspezifischen Daten aus Abschnitt 4.2 sollen nun die Tabellenstrukturen des Referenzmodells schrittweise befüllt werden, um darüber das Funktionsprinzip zu verdeutlichen. Als funktionale Aspekte stehen hierbei wie in Abschnitt 3.2.2 definiert die Sprache, Region, Version und Staffel zur Verfügung. Bevor sich diese im System aufnehmen lassen, sind passende Datentypen in der Tabelle AspectDatatype entsprechend **Abbildung 4.4** zu erfassen.

AspTypeID	TypeName	Length	Scale
101	INTEGER	8	-
102	VARCHAR	5	-
103	VARCHAR	20	-
104	VARCHAR	255	-
105	NUMERIC	8	2

Abbildung 4.4: Beispieldaten in Tabelle AspectDatatype

Anschließend können wie in **Abbildung 4.5** skizziert die erwähnten Aspekte in der Tabelle AspectDefinition hinterlegt werden.

AspDefID	Name	Key	Datatype
201	Sprache	Locale	102
202	Region	Markt	103
203	Version	Nummer	101
204	Staffel	Art	103

Abbildung 4.5: Beispieldaten in Tabelle AspectDefinition

Hierbei spezifiziert der im Attribut DATATYPE referenzierte Wert, welchen Datentyp die Aspektschlüsselwerte zu einem Aspekt besitzen. Trotz prinzipiell frei wählbarer Bezeichnungen der Datentypen orientiert sich das vorliegende Szenario bezüglich Benennung und Semantik an der SQL-Norm. Eine Auswahl typischer sowie im weiteren Verlauf zur Demonstration benötigter Aspektschlüsselwerte mit Verweis auf den jeweiligen funktionalen Aspekt ist in **Abbildung 4.6** dargestellt.

ASPKEYID	ASPECT	KEYVALUE	COMMENT
301	201	en	Britisches Englisch
302	201	fr	Französisch
303	202	UK	United Kingdom
304	202	EU	Europäische Union
305	203	18	Revision zum Jahresende
306	204	Standard	Endkunden-Preis

Abbildung 4.6: Beispieldaten in Tabelle ASPECTKEYVALUE

Damit sind alle Stammdaten zu funktionalen Aspekten angegeben. Im nächsten Schritt muss die Tabelle ASPECTTABLE mit Metadaten zu denjenigen Relationen des fachlichen Modells befüllt werden, auf denen die Aspekte wirken sollen. Da im Anwendungsbeispiel nur die eine Tabelle MODULE existiert, repräsentiert sie auch in **Abbildung 4.7** den einzigen Datensatz.

ASPTABID	SCHEMA	TABLENAME	KEYNAME
401	Test	Module	RowID

Abbildung 4.7: Beispieldaten in Tabelle ASPECTTABLE

Da konkrete Aspektabhängigkeiten auf Attributen einer Relation zu definieren sind, müssen dafür in der Tabelle ASPECTCOLUMN alle relevanten, d.h. gemäß Abbildung 3.5 aspektspezifischen, Attribute registriert werden. **Abbildung 4.8** verdeutlicht das Ergebnis, wobei der angegebene Datentyp jeweils eine Fremdschlüssel-Referenz auf einen Eintrag in der zu Beginn befüllten Tabelle ASPECTDATATYPE verkörpert.

ASPCOLID	TABLE	COLUMNNAME	DATATYPE
501	401	Name	103
502	401	Description	104
503	401	Material	103
504	401	Price	105
505	401	Norm	103

Abbildung 4.8: Beispieldaten in Tabelle ASPECTCOLUMN

Auf dieser Grundlage können nun die Abhängigkeiten zwischen einem Attribut und den darauf wirkenden Aspekten in der Tabelle ASPECTDEPENDENCE gespeichert werden. Dazu ist für jedes in Abbildung 3.5 angegebene Attribut je zugeordnetem Aspekt ein entsprechender Datensatz mit den Fremdschlüssel-Verweisen auf die Katalog-Tabellen ASPECT-

Column und AspectDefinition anzulegen. **Abbildung 4.9** zeigt das Ergebnis dieses Prozesses. Zur besseren Übersicht sind darin die zusammengehörenden Zeilen gruppiert, wobei deren Bedeutung mit Hilfe der Abhängigkeitsfunktion Δ gemäß Definition 4.2 formalisiert wurde.

Column	Aspect	
501	201	} Δ(Name) = {Sprache, Version}
501	203	
502	201	} Δ(Description) = {Sprache, Version}
502	203	
503	201	} Δ(Material) = {Sprache, Version}
503	203	
504	202	} Δ(Price) = {Region, Version, Staffel}
504	203	
504	204	
505	202	} Δ(Norm) = {Region, Version}
505	203	

Abbildung 4.9: Beispieldaten in Tabelle AspectDependence

Nachdem die Tabellen des Aspektkatalogs alle notwendigen Metadaten enthalten, lassen sich nun die eigentlichen aspektspezifischen Daten erfassen. Diese müssen gemäß der Abgrenzung in Abschnitt 4.1 einen Bezug zu existierenden Attributwerten im fachlichen Modell haben. Aus diesem Grund sei beispielhaft der Datensatz mit $RowID = 3141$ in der Tabelle Module betrachtet:

$$(\textit{'mod3141'}, \textit{'Schraube'}, 0.37, \textit{'Innensechskant'}, \textit{'DIN 911'},\\ \textit{'Edelstahl'}, \textit{'Matthias Liebisch'}, 20120823, 3141)$$

Zu diesem fachlichen Kontext können die in Definition 4.4 verwendeten Beispielwerte in der Tabelle AspectValue persistiert werden, **Abbildung 4.10** veranschaulicht das Resultat.

AspValID	Table	Column	RowID	Value	Wert ist zugeordnet ...
701	401	501	3141	screw	} Module.Name
702	401	501	3141	vis	
703	401	504	3141	0.33	} Module.Price
704	401	504	3141	0.37	
705	401	502	3141	hex socket	} Module.Description
706	401	502	3141	à six pans creux	
707	401	505	3141	BS 4190	} Module.Norm
708	401	505	3141	ISO 4016	
709	401	503	3141	stainless steel	} Module.Material
710	401	503	3141	inox	

Abbildung 4.10: Beispieldaten in Tabelle AspectValue

Abschließend ist noch in der Tabelle AspectAssign die Zuordnung zwischen aspektspezifischen Werten und Aspektschlüsselwerten zu treffen. Um das signierte Tupel

$$\sigma[\textit{'screw'}, 0.33, \textit{'hex socket'}, \textit{'BS 4190'}, \textit{'stainless steel'}]_R$$

mit dem Relationenschema $R = (\text{Name}, \text{Price}, \text{Description}, \text{Norm}, \text{Material})$ und der Signatur $\sigma = (en, UK, 18, \textit{Standard})$ darstellen zu können, sind die in **Abbildung 4.11** aufgeführten Tupel notwendig. Im Sinne einer besseren Übersicht wurden die zusammengehörenden Zeilen gruppiert und mit ihrer Aspektsignatur gemäß Definition 4.4 versehen.

AspAssID	KeyValue	Aspect	AspectValue	
801	301	201	701	$(en, 18)[\textit{'screw'}]_{(\text{Name})}$
802	305	203	701	
803	303	202	703	$(UK, 18, \text{Standard})[0.33]_{(\text{Price})}$
804	305	203	703	
805	306	204	703	
806	301	201	705	$(en, 18)[\textit{'hex socket'}]_{(\text{Description})}$
807	305	203	705	
808	303	202	707	$(UK, 18)[\textit{'BS 4190'}]_{(\text{Norm})}$
809	305	203	707	
810	301	201	709	$(en, 18)[\textit{'stainless steel'}]_{(\text{Material})}$
811	305	203	709	

Abbildung 4.11: Beispieldaten in Tabelle AspectAssign

4.4 Zugriffsmodell

Nach Beschreibung der Speicherstrukturen steht nun der Zugriff innerhalb des Referenzmodells im Fokus der Betrachtungen, um die deskriptive Verarbeitung vollständiger aspektspezifischer Tupel durch eine Anwendung unterstützen zu können. Dazu werden einleitend die formal notwendigen Transformationen in Abschnitt 4.4.1 betrachtet. Daraufhin erfolgt in Abschnitt 4.4.2 die Analyse potentieller Zugriffstechniken, bevor diese abschließend in Abschnitt 4.4.3 bewertet und verglichen werden.

4.4.1 Transformation

Dieser Abschnitt erläutert die Zusammenhänge zwischen dem Persistenz- und Zugriffsmodell. Konkret erfordert dies die Überführung aspektspezifischer Daten aus den Tabellen AspectValue und AspectAssign in signierte Tupel gemäß Abschnitt 4.2.2 und umgekehrt. Zur Veranschaulichung dient folgendes gegenüber dem Beispiel in Definition 4.4 leicht modifiziertes signiertes Tupel.

$$(en, UK, 18)[\textit{'screw'}, \textit{'hex socket'}, \textit{'BS 4190'}]_{(\text{Name},\text{Description},\text{Norm})}$$

Dabei sind die Attribute Name und Description wie in Abbildung 3.5 skizziert jeweils von den Aspekten Sprache (en) und Version (18) abhängig, während Norm von den Aspekten Region (UK) und Version (18) beeinflusst wird.

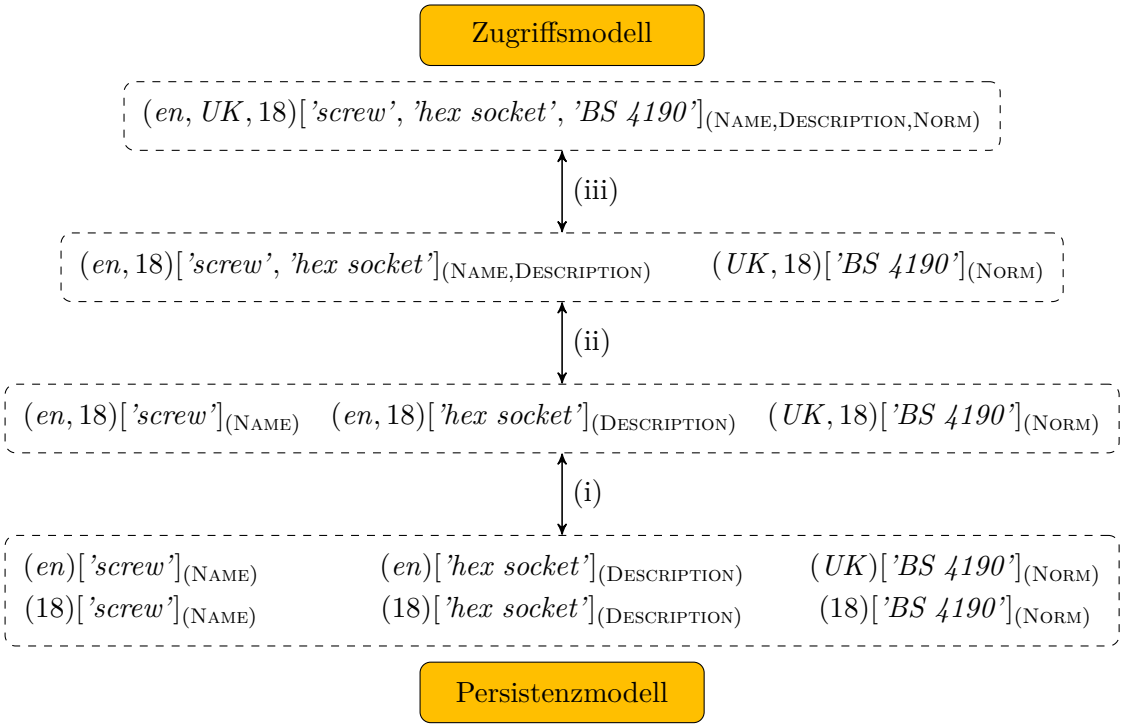


Abbildung 4.12: Transformation zwischen Zugriffs- und Persistenzmodell nach [Pie11b]

Abhängig von der Art des Zugriffs (lesend, schreibend) lassen sich wie in **Abbildung 4.12** dargestellt zwei unterschiedliche Transformationsrichtungen identifizieren, die nachfolgend genauer betrachtet werden sollen.

4.4.1.1 Lesender Zugriff

Grundlage für die Beantwortung lesender Anfragen zu aspektspezifischen Daten ist die Tabelle AspectAssign in der untersten Ebene, wobei jedes Tupel wie beispielsweise $(18)['screw']_{(\text{Name})}$ einen Eintrag in jener Tabelle darstellt. Da jedoch die eigentlichen Werte durch die Tabelle AspectValue nach dem EAV-Prinzip gespeichert werden, ist im Schritt (i) eine sogenannte **Pivotisierung** [CGGL04] bezüglich der Attributwerte erforderlich. Hieraus resultieren die signierten Attribute in der darüber liegenden Ebene. Eine zweite Pivotisierung nach den Signaturen in Schritt (ii) erzeugt auf der vorletzten Ebene uniforme Tupel (siehe Definition 4.9). Schließlich ergibt sich mit Schritt (iii) das angefragte signierte Tupel gemäß der in Definition 4.12 beschriebenen Tupel-Ableitung.

4.4.1.2 Schreibender Zugriff

Die Persistierung von zu ändernden Daten aus der Zugriffsebene erfolgt durch Umkehrung der zuvor beschriebenen Schrittfolge und beginnt mit einer uniformen Zerlegung im Schritt (iii) als Invertierung der Tupel-Ableitung. Die folgenden Pivotisierungen sowohl bezüglich der Signaturen im Schritt (ii) als auch nach den Attributwerten im Schritt (i) sind nach dem Kriterium von Wyss und Robertson [WR05] vollständig umkehrbar, wie durch einen formalen Beweis in [Pie11a] gezeigt.

4.4.2 Analyse von Techniken

Dieser Abschnitt dient der Vorstellung geeigneter Zugriffstechniken basierend auf [Lie11a] für die Auswertung der in Abschnitt 4.3.2 definierten Strukturen. Zum besseren Verständnis soll deren Funktionsweise anhand der folgenden Beispiel-Anfrage auf der in Abschnitt 3.2 angegebenen Relation MODULE erläutert werden.

> *Ermittle für das Modul mit dem Schlüssel 'mod3141' alle mehrsprachigen, d.h. vom Aspekt 'Sprache' beeinflussten, Daten für die Attribute* DESCRIPTION *und* NAME *sowie den jeweils zugeordneten Aspektschlüsselwert für die Locale.*

Die zwei Tupel *('screw','hex socket','en')* und *('vis','à six pans creux','fr')* stellen das erwartete Ergebnis bezüglich der Beispieldaten in Abschnitt 4.3.3 dar.

4.4.2.1 SQL mit JOIN

Unter Beschränkung auf den SQL:92-Standard [ISO92] gemäß Universalitäts-Anforderung im AOD-Paradigma ist die Transformation wie in Abschnitt 4.4.1 beschrieben nur mit Hilfe des JOIN-Operators durchführbar. Das prinzipielle Vorgehen ist anhand obiger Beispielanfrage in **Abbildung 4.13** dargestellt. Hierbei wurde vorausgesetzt, dass Idents von Aspekten und Tabellenspalten als Metadaten bei der Anfrageerstellung bekannt sind.

```
SELECT T1.Value AS Name, T2.Value AS Description,
       T5.KeyValue AS Locale
  FROM AspectValue T1
     INNER JOIN AspectValue T2 ON T1.RowID = T2.RowID
     INNER JOIN AspectAssign T3 ON T1.AspValID = T3.AspectValue
     INNER JOIN AspectAssign T4 ON T2.AspValID = T4.AspectValue
     INNER JOIN AspectKeyValue T5 ON T3.KeyValue = T5.AspKeyID
WHERE T1.RowID = 3141            -- /* entspricht Key='mod3141' */
  AND T1.Column = 501            -- /* Name */
  AND T2.Column = 502            -- /* Description */
  AND T3.KeyValue = T4.KeyValue  -- /* keine Locales vermischen */
  AND T5.Aspect = 201            -- /* Sprach-Aspekt */
```

Abbildung 4.13: Referenzmodell-Anfrage mit JOIN

Trotz einer sehr einfachen Beispielanfrage ist für deren Beantwortung bereits eine recht umfangreiche SQL-Struktur zu formulieren. Die Ursache hierfür liegt in der Pivotisierung nach Aspekt-Signaturen mit Hilfe des JOIN-Operators aufgrund fehlender Alternativen. Wie in Abbildung 4.13 zu erkennen, muss ein solcher JOIN basierend auf Tabelle ASPECTVALUE für jedes im Ergebnisschema anzuzeigende Attribut spezifiziert werden, d.h. die Anzahl der JOIN-Operatoren (vier im obigen Beispiel) skaliert linear mit den angefragten Attributen. Hierbei ist jeweils eine weitere Verknüpfung zwischen ASPECTVALUE und ASPECTASSIGN notwendig, welche typischerweise die mit Abstand größten Kardinalitäten im gesamten Referenzmodell aufweisen. Dementsprechend haben derartige Anfragen häufig eine inakzeptable Antwortzeit, wie Analysen in [DNB06, Lie11b, LP10] zeigen.

4.4.2.2 SQL mit PIVOT

Für eine effizientere Formulierung und Verarbeitung der im Referenzmodell notwendigen Pivotisierungs-Anfragen wird statt des allgemeinen JOIN-Operators ein dediziertes Sprachkonstrukt benötigt. Da jedoch selbst in der aktuellen SQL:2011-Norm [ISO11] kein solcher PIVOT-Operator definiert ist, muss bei Verwendung DBMS-spezifischer Spracherweiterungen die Universalitäts-Anforderung des AOD-Paradigmas fallen gelassen werden. Konkret bieten unter anderem die DBMS-Produkte Microsoft SQL Server 2012 [Mic12] und Oracle 11g [Ora12] Implementierungen des PIVOT- und UNPIVOT-Operators. Deren ursprüngliches Anwendungsgebiet liegt im Bereich Data Warehouse für die Transformation von OLAP-Anfragen [RI09] zur inhaltlichen Veränderung des Blickwinkels auf die Daten, beispielsweise um eine Umsatzübersicht nach Regionen statt Produkten auszurichten. Darüber hinaus ist der PIVOT-Operator für die Auswertung von EAV-Tabellen spezialisiert, dessen Anwendung im Referenzmodell ist in **Abbildung 4.14** anhand der Beispielanfrage auf Basis des Microsoft SQL Server 2012 illustriert.

```
SELECT PivotedData.[1] AS Name, PivotedData.[2] AS Description,
       PivotedData.KeyValue AS Locale
  FROM
  (
    SELECT T1.RowID, T1.Column, T1.Value, T3.KeyValue
      FROM AspectValue T1
         INNER JOIN AspectAssign T2 ON T1.AspValID = T2.AspectValue
         INNER JOIN AspectKeyValue T3 ON T2.KeyValue = T3.AspKeyID
     WHERE T3.Aspect = 201          -- /* Sprach-Aspekt */
       AND T1.RowID = 3141          -- /* entspricht Key='mod3141' */
  ) AS JoinData
  PIVOT
  (
    MAX(JoinData.Value)
    FOR JoinData.Column IN ([501], [502])        -- /* Column-IDs */
  ) AS PivotedData
```

Abbildung 4.14: Referenzmodell-Anfrage mit PIVOT

Effektiv werden durch die innere `SELECT-FROM-WHERE`-Klausel alle aspektspezifischen Werte inklusive des zugehörigen Attributs und der Aspektschlüsselwerte zu einem Datensatz (in der Tabelle MODULE) mit der RowID 3141 und dem Aspekt Sprache ermittelt. Auf diesem als `JoinData` bezeichneten temporären Ergebnis erfolgt die Pivotisierung nach dem Attribut `JoinData.Column` mit der notwendigen Aggregation mittels Maximum-Bildung über die jeweiligen aspektspezifischen Werte. Dieser auf den ersten Blick irritierende Ausdruck belässt das Ergebnis jedoch inhaltlich korrekt aufgrund der Bedingung C5 in Abschnitt 4.3.2.9. Hierbei ist zu beachten, dass für die IN-Klausel des PIVOT-Operators nur definierte Werte zulässig sind. Eine dynamische zur Ausführungszeit ermittelte Wertemenge, beispielsweise durch die Unterabfrage `SELECT Column FROM AspectValue`, lässt sich nur über Stored Procedures bzw. Anwendungscode realisieren. Wie in [CGGL04] dargestellt, erlaubt erst der explizite Einsatz des PIVOT-Operators dessen gezielte Optimierung durch das DBMS auf Basis der klassischen Operatoren Verbund, Selektion und Projektion.

4.4.2.3 SQL mit Spracherweiterung

Eine weitere Alternative gegenüber dem proprietären und ebenfalls komplexen PIVOT-Operator besteht in der Erweiterung von SQL um Sprachkonstrukte, welche einen adäquaten Umgang mit funktionalen Aspekten im Referenzmodell erlauben. Dies betrifft einerseits den DDL-Teil, um beispielsweise ein Attribut einem funktionalen Aspekt zuordnen zu können. Andererseits muss auch der DML-Teil entsprechend erweitert werden, wie dies stellvertretend anhand der in **Abbildung 4.15** für das SELECT-Statement relevanten Tabellenreferenz dargestellt ist.

```
<table reference> ::=
   <table name> [ <correlation specification> ]
 | <derived table> <correlation specification>
 | <joined table>
 | <aspect view>

<aspect view> ::=
   ASPECTVIEW <identifier> BASED ON <table name>
   GROUP BY <aspect key> [{, <aspect key>} ... ]

<aspect key> ::=
   ASPECTKEY(<character string literal>)
   AS <identifier>
```

Abbildung 4.15: Definition der SQL-Erweiterung ASPECTVIEW

Ausgehend von der SQL:92-Norm [ISO92] wird eine neue Alternative `<aspect view>` für das Nichtterminalsymbol `<table reference>` eingeführt. Diese erzeugt durch das neue Schlüsselwort `ASPECTVIEW` eine Sicht auf alle aspektspezifischen Daten zu der über `BASED ON` festgelegten fachlichen Basistabelle, wobei die Informationen bezüglich des in Abschnitt 4.3.2.9 geforderten ein-elementigen Schlüsselattributs und der über `<aspect key>` angegebenen funktionalen Aspekte gruppiert werden. Dadurch ergibt sich für eine Relation R mit n Attributen und insgesamt k zugeordneten Aspekten das nachfolgende virtuelle Relationenschema.

$$R' = (\underbrace{Att_1, \ldots, Att_n}_{Basistabelle}, \underbrace{Asp_1, \ldots, Asp_k}_{Aspekte})$$

Voraussetzung hierfür wäre eine weitere UNIQUE-Bedingung im Referenzmodell auf dem Attribut KEY in der Tabelle ASPECTDEFINITION, um dessen Werte über den Ausdruck `ASPECTKEY(<character string literal>)` eindeutig als Aspektschlüssel referenzieren zu können.

Durch Verwendung des neuen SQL-Sprachkonstrukts `ASPECTVIEW` zur Realisierung der anfänglichen Beispiel-Anfrage fällt diese gegenüber den bisher betrachteten Zugriffstechniken sehr kompakt und leicht verständlich aus. Wie in **Abbildung 4.16** skizziert, wird für die Tabelle MODULE eine Aspektsicht bezüglich des Aspektschlüssels 'Locale' erzeugt, über den gemäß Abbildung 4.5 die Schlüsselwerte des Aspekts Sprache referenziert werden können. Um jene konkreten Locale-Werte für einen aspektspezifischen Datensatz wie gefordert im Ergebnis auszugeben, muss die Tabelle ASPECTKEYVALUE über einen JOIN mit dem temporären Ergebnis `ModuleAspects` (Alias T1) verknüpft werden.

```
SELECT T1.Name, T1.Description, T2.KeyValue AS Locale
  FROM AspectKeyValue T2 INNER JOIN
       ( ASPECTVIEW ModuleAspects
         BASED ON Test.Module
         GROUP BY ASPECTKEY('Locale') AS AspLoc
       ) T1
       ON T2.AspKeyID = T1.AspLoc
 WHERE T1.Key = 'mod3141'
   AND T2.Aspect = 201          -- /* Sprach-Aspekt */
```

Abbildung 4.16: Referenzmodell-Anfrage mit ASPECTVIEW

Die Erfahrungen mit ähnlichen Ansätzen zu Spracherweiterungen von SQL [Lie03, Mül08] zeigen allerdings, dass eine Aufnahme in die Norm oder Umsetzung in DBMS-Produkten so gut wie ausgeschlossen ist. Eine mögliche Lösung stellen sogenannte Proxies [ALB10] oder Query Transformation Layer [AGJ+08] zwischen Anwendungssystem und DBMS dar. Dadurch könnten alle in Abbildung 4.15 neu definierten SQL-Konstrukte durch einen Parser erkannt und in möglichst effiziente produktspezifische SQL-Anweisungen transformiert werden.

4.4.2.4 API-Zugriffsschicht

Im Gegensatz zu den bisherigen auf SQL-Mitteln basierenden Alternativen wird hier ein Ansatz zur anwendungsseitigen Verarbeitung aspektspezifischer Daten vorgestellt. Analog zum Konzept von ODBC als eine einheitliche Schnittstelle für den Zugriff auf relationale Daten soll diese Funktion im Zusammenhang mit aspektspezifischen Daten des Referenzmodells durch einen geeigneten Funktionsbaustein mit entsprechendem Application Programming Interface (API) realisiert werden. Die grundsätzliche Architektur ist in **Abbildung 4.17** dargestellt.

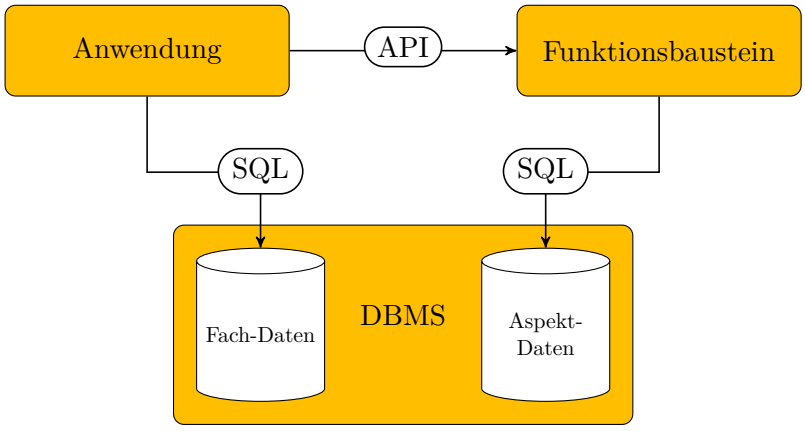


Abbildung 4.17: Referenzmodell-Anfrage mit API-Zugriffsschicht

Durch die Unterscheidung der Zugriffswege auf fachliche Daten per SQL und aspektspezifische Daten über die API besteht im Anwendungssystem die Aufgabe, die Ergebnisse beider Datenquellen soweit erforderlich integriert auszuwerten. Zur Gewährleistung eines möglichst plattformunabhängigen und universellen Funktionsbausteins ist dieser für die in

Kapitel 5 beschriebene Referenzimplementierung in Java realisiert. Dabei wird sowohl die Verwaltung der funktionalen Aspekte als auch die Abfrage und Manipulation aspektspezifischer Daten unterstützt. Beispiele hierfür sowie zahlreiche Implementierungs-Details folgen an entsprechender Stelle.

4.4.3 Vergleich der Techniken

Zum Abschluss dieses Kapitels sollen die hier vorgestellten Zugriffstechniken angelehnt an [Lie11a] verglichen und bewertet werden. Zuvor findet eine Beschreibung der hierfür notwendigen Kriterien statt.

4.4.3.1 Kriterien

Anhand der folgenden Merkmale erfolgt die Bewertung der Zugriffstechniken.

Standardisierung: Ausgehend vom AOD-Paradigma repräsentiert das Merkmal der Standardisierung die Anforderung der Universalität (siehe Abschnitt 3.3), hier jedoch bezüglich einer ISO-Norm.

Praxisrelevanz: Häufig beeinflusst durch die Standardisierung soll das Merkmal der Praxisrelevanz klären, inwieweit eine Zugriffstechnik tatsächlich für den praktischen Einsatz geeignet ist.

Performance: Als das wohl wichtigste Merkmal ist letztendlich die Performance eines Systems entscheidend. Hierzu soll vorerst eine qualitative Abschätzung als Bewertungsmaßstab dienen, zum Abschluss der vorliegenden Arbeit werden in Kapitel 6 konkrete Messwerte eines Benchmarks vorgestellt.

Transparenz: In der Transparenz einer Zugriffstechnik zeigt sich, in welchem Ausmaß ein Anwender Kenntnis zu den Strukturen des Referenzmodells im Rahmen des Zugriffs auf aspektspezifische Daten besitzen muss.

Funktionsadäquatheit: Das Kriterium der Funktionsadäquatheit gibt an, ob der betreffende Ansatz auf die Mächtigkeit des zugrunde liegenden RDBMS zurückgreift oder stattdessen Teil-Funktionalität wie das Cachen von Daten reimplementiert.

Erweiterbarkeit: Ebenfalls abhängig von der Standardisierung bewertet das Merkmal der Erweiterbarkeit die potentiellen Möglichkeiten zur Anpassung des Funktionsumfangs einer Zugriffstechnik.

Benutzerfreundlichkeit: Entsprechend der gleichnamigen Anforderung zum AOD-Paradigma in Abschnitt 3.3 soll das Merkmal der Benutzerfreundlichkeit beurteilen, wie handlich und intuitiv sich jeweils eine Zugriffstechnik zur Abfrage aspektspezifischer Daten darstellt.

4.4.3.2 Bewertung und Fazit

Nachfolgend werden die vier in Abschnitt 4.4.2 erläuterten Zugriffstechniken bezüglich der soeben aufgestellten Kriterien bewertet, **Tabelle 4.1** bietet diesbezüglich eine Zusammenfassung der Ergebnisse.

Kriterium	JOIN	PIVOT	VIEW	API
Standardisierung	+	o[1]	–	o[2]
Praxisrelevanz	+	o[1]	o[3]	+
Performance	–	o	o	+[4]
Transparenz	–	o	+	+
Funktionsadäquatheit	+	+	o	–
Erweiterbarkeit	–	–	+	+
Benutzerfreundlichkeit	–	o	+	o

+: ja/hoch o: neutral/mittel –: nein/niedrig

[1] unterstützt durch Oracle 11g und MS SQL Server 2005
[2] de-facto Standard [Egy01] aufgrund enormer Verbreitung
[3] nur bei Nutzung von Zwischenschichten/Proxies
[4] begründet durch Ergebnisse in [DNB06] und Kapitel 6

Tabelle 4.1: Bewertung der Zugriffstechniken für das Referenzmodell

Der **JOIN**-Ansatz greift zwar als einziger Vertreter auf standardisierte Sprachkonstrukte zurück und besitzt damit eine große Praxisrelevanz durch eine breite Produktunterstützung, ist allerdings weder performant noch benutzerspezifisch erweiterbar. Zudem fehlt die Benutzerfreundlichkeit und Transparenz aufgrund notwendiger Kenntnisse zum Aufbau der Aspekt-Tabellen im Referenzmodell.

Demgegenüber verspricht die **PIVOT**-Variante einen etwas transparenteren Zugang zu aspektspezifischen Daten sowie eine bessere Performance bei deren Verarbeitung. Allerdings gibt es auch hier keine Möglichkeit zur Erweiterung und die Nutzung des (noch) nicht standardisierten Konstrukts ist auf bestimmte DBMS-Hersteller beschränkt, wodurch sich eine geringere Praxisrelevanz ergibt.

Trotz fehlender Standardisierung der **ASPECTVIEW**-Technik in der SQL-Norm lässt sich diese durch Verwendung von zusätzlichen Zwischenschichten oder Proxies [ALB10] in der Praxis einsetzen. Da jedoch das Transformations-Ergebnis immer nur auf bekannte SQL-Konstrukte führen kann, bestimmt deren Ausführungsgeschwindigkeit letztendlich auch die Performance. Andererseits steht dem Anwendungsentwickler ein intuitives und durch die fehlende Standardisierung erweiterbares Konzept zur Verfügung, um transparent auf aspektspezifische Daten zugreifen zu können.

Der im **API**-Ansatz zugrunde liegende Java-Stack besitzt zwar eine Standardisierung [Egy01], allerdings ist die Schnittstelle selbst proprietär. Aufgrund der enormen Verbreitung von Java ist allerdings die Praxisrelevanz hoch. Wie die Ausführung in Kapitel 6 zeigen werden, lassen sich über dieser Schnittstelle deutlich bessere Antwortzeiten erreichen. Zudem bieten die Funktionen der API eine gute Kapselung aller Strukturen im Referenzmodell gegenüber dem Nutzer. Die prinzipiell gute Erweiterbarkeit unterliegt jedoch dem Problem, die Mächtigkeit und Komplexität von SQL adäquat abzubilden.

Unter Beachtung aller Kriterien stellt das API-Konzept somit die am besten geeignete Variante für den Zugriff auf das Referenzmodell dar. Im folgenden Kapitel 5 wird diesbezüglich eine prototypische Realisierung präsentiert.

Kapitel 5

Prototypische Implementierung

Wie die vorangegangene Bewertung der Zugriffstechniken gezeigt hat, stellt eine Zugriffsschicht die erfolgversprechendste Variante zur Verarbeitung komplexer aspektspezifischer Daten im Referenzmodell des AOD-Paradigmas dar. Dieses Kapitel dient nun dem Nachweis der Funktionsfähigkeit und Praxistauglichkeit einer solchen API, indem eine auf [Pie11a, Pie11b] basierende prototypische Realisierung vorgestellt wird. Dazu müssen einleitend in **Abschnitt 5.1** grundlegende Fragestellungen analysiert und Designentscheidungen festgelegt werden, welche beim Einsatz einer Zugriffsschicht zu berücksichtigen sind. Daraufhin erfolgt in **Abschnitt 5.2** die Beschreibung zentraler Strukturen und Methoden der eigentlichen API. Abschließend sollen einige Beispiele in **Abschnitt 5.3** deren Verwendung ansatzweise demonstrieren.

5.1 Vorüberlegungen

Für die Implementierung der Zugriffsschicht ist im Vorfeld die Klärung zentraler konzeptueller Details hinsichtlich Verwendung und Wirkungsweise der API notwendig. Sowohl mit dem in Abschnitt 5.1.1 diskutierten Integrationsgrad zwischen dem DBMS und der eigentlichen Anwendung als auch durch die Beschreibung von Möglichkeiten zur Filterung in Abschnitt 5.1.2 soll dabei die angestrebte Nutzung der Zugriffsschicht genauer spezifiziert werden. Deren konkrete Arbeitsweise mit Fokussierung auf die erforderlichen Datentransformationen wird anschließend in Abschnitt 5.1.3 erläutert.

5.1.1 Einbindung der Zugriffsschicht

Ausgehend von einer Vielzahl an Applikationen, welche über ein zentrales DBMS auf eine relationale Datenbank und deren aspektspezifische Daten zugreifen möchten, gibt es für die konkrete Ausprägung einer vermittelnden Zugriffsschicht verschiedene charakterisierende Merkmalsdimensionen. Wie in **Abbildung 5.1** dargestellt, betrifft dies einerseits die Art der Kopplung sowohl zum DBMS als auch bezüglich der Applikationen. Andererseits stellt sich die Frage nach dem Grad der System-Einbettung sowie der Fähigkeit zur Zwischenspeicherung. Nach einer näheren Betrachtung dieser Merkmale wird abschließend das zur prototypischen Implementierung der API genutzte Szenario vorgestellt.

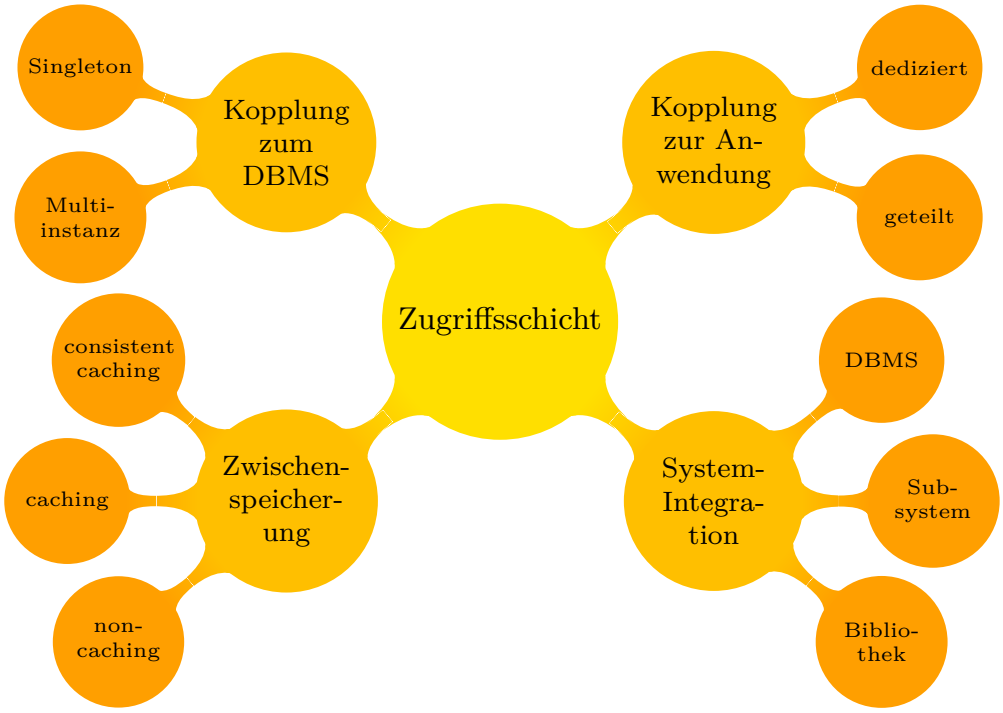


Abbildung 5.1: Klassifikation zur Charakterisierung der Zugriffsschicht

5.1.1.1 Kopplung zum DBMS

Für die Interaktion zwischen Zugriffsschicht und Datenbanksystem gibt es zwei prinzipielle Möglichkeiten.

Singleton: Die Verwaltung der aspektspezifischen Informationen in der Datenbank erfolgt in genau einer Instanz der Zugriffsschicht, über deren API sich demzufolge alle Anwendungen verbinden müssen. Durch jene Zentralisierung ist kein Zusatzaufwand für die Konsistenzsicherung im Fall eines Zwischenspeichers notwendig. Das damit verbundene Potential einer hohen Performance geht allerdings zu Lasten der Skalierbarkeit bei einer wachsenden Zahl von Anwendungen.

Multiinstanz: Zu einer Datenbank existieren mehrere Instanzen der Zugriffsschicht, wobei die Zuordnung zu einer konkreten Anwendung nach gewissen Kriterien erfolgt. Im Sinne einer klassischen Lastverteilung kann dadurch eine sehr gute Skalierbarkeit erreicht werden. Allerdings müssen diese Instanzen untereinander koordiniert werden, was entweder implizit durch das DBMS mittels Transaktions- und Sperrmechanismen erfolgen kann oder im Entwurf einer Synchronisations-Komponente zu berücksichtigen ist.

5.1.1.2 Kopplung zur Anwendung

Analog zur Anbindung des DBMS lassen sich auch zwei Varianten für die Kopplung von Anwendungen an Zugriffsschicht-Instanzen unterscheiden.

dediziert: Jede Anwendung erzeugt sich eine eigene exklusive Instanz der Zugriffsschicht, sobald zur Verarbeitung aspektspezifischer Informationen eine entsprechende Datenbank-Verbindung benötigt wird. Deren Lebensdauer erstreckt sich damit bis zum Beenden der übergeordneten Applikation.

geteilt: Einer Zugriffsschicht-Instanz können verschiedene Anwendungen zugeordnet sein. Die hierbei notwendige Erzeugung, Zuweisung und Löschung von Instanzen muss durch eine zusätzliche Lastverteilungs-Komponente realisiert werden.

5.1.1.3 System-Integration

Für einen benutzerfreundlichen Umgang mit aspektspezifischen Daten im Sinne des AOD-Paradigmas bieten sich für die Zugriffsschicht drei Alternativen zur Integration in eine Systemlandschaft.

DBMS: Auf Basis von Stored Procedures und erweiterter Funktionen kann eine vollständige Einbettung der Zwischenschicht in das DBMS erreicht werden. Dadurch sind sowohl API-Aufrufe über standardisiertes SQL als auch die Vermischung aspektrelevanter und aspektunabhängiger Zugriffe in einer Anfrage möglich. Allerdings wird durch eine je DBMS-Produkt spezifische Implementierung das AOD-Paradigma bezüglich der Anforderung zur Universalität nicht erfüllt.

Subsystem: Die Zugriffsschicht ist als autarke Komponente in der gesamten Systemarchitektur konzipiert und interagiert mit dem DBMS über standardisierte Protokolle wie beispielsweise ODBC. Dabei ist deren physische Nähe zum Datenbankserver ausschlaggebend für den möglichst geringen zusätzlichen Kommunikationsaufwand sowie die potentielle Zwischenspeicherung unter Nutzung gemeinsamer Ressourcen.

Bibliothek: Mit Einbindung der Zugriffsschicht als Anwendungs-Bibliothek ergibt sich schließlich die maximale Entkopplung von der ursprünglichen Datenbank. Dadurch sollte das Kommunikationsprotokoll zum DBMS so schlank wie möglich gehalten werden. Andererseits vereinfachen sich Aufbau und Design der Zugriffsschicht, da jede Instanz nur noch für die Verwaltung von Daten und Anfragen der übergeordneten Anwendung zuständig ist.

5.1.1.4 Zwischenspeicherung

Zur Steigerung der Performance können auch in der Zugriffsschicht gewisse Strategien zur Zwischenspeicherung aspektspezifischer Daten angewendet werden. Dabei sind die Auswirkungen auf die Konsistenz insbesondere bei Multiinstanzen zu berücksichtigen.

non-caching: Es findet keine Zwischenspeicherung statt, jeder Zugriff auf aspektspezifische Daten erfordert eine Kommunikation mit dem DBMS.

caching: Die Zugriffsschicht verwaltet optimierte Datenstrukturen, in denen aspektspezifische Daten in geeigneter Form zwischengespeichert werden. Dadurch reduziert sich sowohl der Kommunikationsaufwand zum DBMS als auch die Prozessorlast für die Transformation von Datenbank-Anfrageergebnissen.

consistent caching: Analog zum vorherigen Fall nimmt auch hier die Zugriffsschicht eine Zwischenspeicherung der Daten vor. Darüber hinaus wird unter zusätzlichem Kommunikationsaufwand für Konsistenz gegenüber der Datenbank im Sinne der ACID-Kriterien [HR01] sowie eventuell weiterer Instanzen der Zugriffsschicht gesorgt.

5.1.1.5 Fazit

Die zuvor erläuterten Ausprägungen der Merkmalsklassifikation lassen sich nicht beliebig miteinander kombinieren, manche bedingen einander und andere schließen sich gegenseitig aus. Beispielsweise ist die Zugriffsschicht per Definition nicht gleichzeitig als Singleton-Kopplung zum DBMS und dediziert hinsichtlich der Anwendungs-Kopplung realisierbar.

Für den weiteren Verlauf des Kapitels wird stattdessen die grundlegendste Variante aller Zugriffsschicht-Implementierungen festgelegt. Diese besteht aus einer Bibliothek, welche durch eine Anwendung instanziiert werden kann, jener dann dediziert zur Verfügung steht und alle notwendigen Datentransformationen bezüglich aspektspezifischer Daten durchführt. Auf die Koordination der dabei erzeugten Zugriffsschicht-Instanzen gegenüber dem Datenbanksystem wird hierbei allerdings ebenso verzichtet wie auf die Zwischenspeicherung von Daten. Das resultierende Szenario bezüglich Anwendung, Zugriffsschicht und DBMS ist in **Abbildung 5.2** schematisch dargestellt.

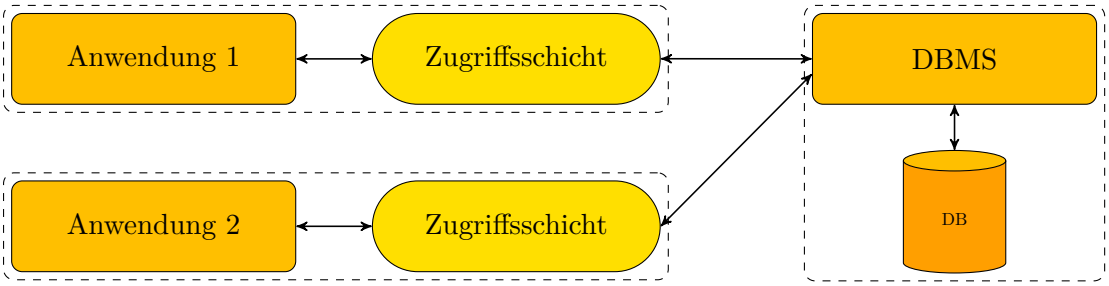


Abbildung 5.2: Architektur-Schema zur Einbindung der Zugriffsschicht

5.1.2 Filtermöglichkeiten der Zugriffsschicht

Die Filterung der Ergebnismenge ist ein zentrales Element zur Spezifikation einer Datenbank-Anfrage. Im Rahmen der AOD sind hierbei zusätzliche Angaben zu relevanten Aspekten und Aspektschlüsselwerten notwendig. Darüber hinaus sollte die Zugriffsschicht ebenso die Möglichkeit bieten, Anfragen hinsichtlich aspektspezifischer Attributwerte einzuschränken. Entsprechende Konzepte für eine Anfragesprache werden in den beiden folgenden Abschnitten vorgestellt.

5.1.2.1 Infix-Aspektfilter

Zur Festlegung von Aspektzugehörigkeiten sind für das Referenzmodell die Aspektfilter AF gemäß Abschnitt 4.2.4 definiert. Diesen fehlt jedoch aufgrund ihrer mathematischen Struktur in Präfixnotation[1] die intuitive Anwendbarkeit. Deshalb werden hier über eine kontextfreie Grammatik die Infix-Aspektfilter als eine funktional identische Sprache

[1] Kennzeichen der Präfixnotation ist die Angabe des Operators vor den Operanden.

vorgestellt, die jedoch mittels der an SQL angelehnten Syntax leichter verständlich und damit nutzerfreundlicher ist. Das Syntaxdiagramm in **Abbildung 5.3** zeigt dabei die Ableitungsmöglichkeiten für einen Infix-Aspektfilter-Ausdruck *expr*.

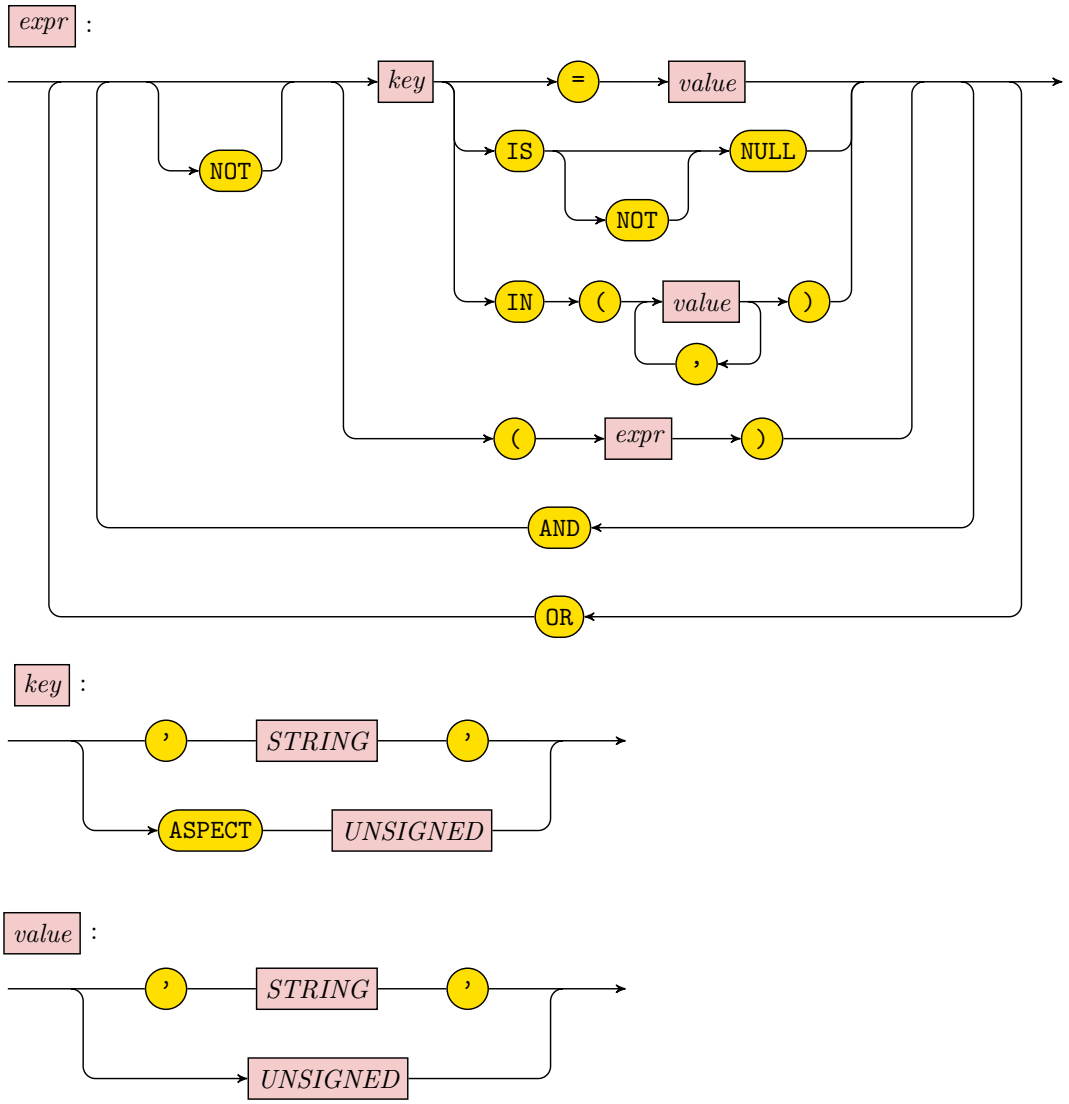


Abbildung 5.3: Syntaxdiagramm für Infix-Aspektfilter nach [Pie11b]

Dabei referenziert *key* den funktionalen Aspekt, welcher einerseits über seine Bezeichnung (*STRING*) oder andererseits mit Hilfe seiner Aspekt-ID (*UNSIGNED*[2]) spezifiziert werden kann. Unter Einbeziehung des in Abschnitt 4.3.2 vorgestellten Referenzmodells entsprechen diese Informationen den Attributen AspectDefinition.Name beziehungsweise AspectDefinition.AspDefID. Analog lässt sich der Aspektschlüsselwert *value* über eine konkrete Ausprägungszeichenkette (*STRING*, AspectKeyValue.KeyValue) oder alternativ durch Angabe der internen Aspektschlüsselwert-ID (*UNSIGNED*, AspectKeyValue.AspKeyID) festlegen.

[2] *UNSIGNED* steht hierbei für eine positive Ganzzahl.

Die Semantik eines solchen Infix-Aspektfilter-Ausdrucks ergibt sich durch schrittweise Überführung in einen *AF*-Ausdruck mit Hilfe der Transformationsvorschrift in Definition 5.1. Die Bedeutung des resultierenden Präfix-Aspektfilters ist dann bezüglich der Ausführungen in Abschnitt 4.2.4 bestimmt.

▶ **Definition 5.1** *Sei $f \in IAF$ ein beliebiger Infix-Aspektfilter-Ausdruck, welcher aus dem Syntaxdiagramm gemäß Abbildung 5.3 ableitbar ist. Dann lässt sich f mit Hilfe einer Abbildung t_{AF}*

$$t_{AF} : IAF \to AF \tag{5.1}$$

in einen Ausdruck der Sprache AF transformieren, wobei nachfolgende Regeln von außen nach innen anzuwenden sind:

$$t_{AF}(f) = \begin{cases} or\,(t_{AF}(a_1), \ldots, t_{AF}(a_n)) & : \textit{falls } f = "a_1 \ OR \ \ldots \ OR \ a_n" \quad (5.2) \\ and\,(t_{AF}(a_1), \ldots, t_{AF}(a_n)) & : \textit{falls } f = "a_1 \ AND \ \ldots \ AND \ a_n" \quad (5.3) \\ not\,(t_{AF}(a_1)) & : \textit{falls } f = "NOT \ a_1" \quad (5.4) \\ t_{AF}(expr) & : \textit{falls } f = "(expr)" \quad (5.5) \\ keyValue(key, value) & : \textit{falls } f = "key = value" \quad (5.6) \\ keyNull(key) & : \textit{falls } f = "key \ IS \ NULL" \quad (5.7) \\ not\,(keyNull(key)) & : \textit{falls } f = "key \ IS \ NOT \ NULL" \quad (5.8) \\ or\,(keyValue(key, v_1), \ldots) & : \textit{falls } f = "key \ IN \ (v_1, \ldots)" \quad (5.9) \end{cases}$$

Bemerkung. Aufgrund der Tatsache, dass für alle Konstrukte des Syntaxdiagramms eine korrelierende Teilregel in t_{AF} existiert, ist diese Transformation vollständig.

Beispiel. Für den Zugriff auf aspektspezifische Daten soll ein Filter vorgegeben werden, der nur explizite Versionszuordnungen, die Sprache *Französisch (fr)* sowie die Marktregionen *Europa (EU)* und die *Schweiz (CH)* zulässt. Diese Anforderungen lassen sich intuitiv aus der natürlich-sprachlichen Formulierung als Infix-Aspektfilter f darstellen, wobei Aspekte und Aspektschlüsselwerte jeweils über ihre Bezeichnungen spezifiziert sind:

$$f = "\textit{'Version' IS NOT NULL AND 'Sprache'='fr' AND 'Markt' IN ('EU','CH')}"$$

Schrittweise und nach dem *Top-Down-Prinzip* kann f basierend auf Definition 5.1 in einen Aspektfilter-Ausdruck mit Präfixschreibweise transformiert werden:

$$\begin{aligned} t_{AF}(f) =_{(5.3)} \ & and\Big(\ t_{AF}("\textit{'Version' IS NOT NULL}"), \\ & \qquad t_{AF}("\textit{'Sprache'='fr'}"), \\ & \qquad t_{AF}("\textit{'Markt' IN ('EU','CH')}")\Big) \\ =_{(5.8),(5.6),(5.9)} \ & and\Big(\ not(keyNull(\textit{'Version'})), \\ & \qquad keyValue(\textit{'Sprache'},\textit{'fr'}), \\ & \qquad or(keyValue(\textit{'Markt'},\textit{'EU'}), keyValue(\textit{'Markt'},\textit{'CH'}))\Big) \end{aligned}$$

5.1.2.2 Wertfilter

Neben der zuvor erläuterten Filterung auf Aspektschlüsselwerte sollte sich eine Anfrage auch bezüglich ihrer Attributwerte einschränken lassen, wie beispielsweise durch die WHERE-Klausel in SQL. Allerdings stellt die Nachbildung ähnlich umfangreicher Wertvergleichsmöglichkeiten für die methodenbasierte API der Zugriffsschicht eine durchaus komplexe Aufgabe dar. Deswegen wurde basierend auf dem in **Abbildung 5.4** skizzierten Syntaxdiagramm eine Wertfilter-Sprache definiert. Diese erlaubt die Formulierung von Vergleichs-Ausdrücken als Zeichenketten und deren Übergabe an die Zugriffsschicht.

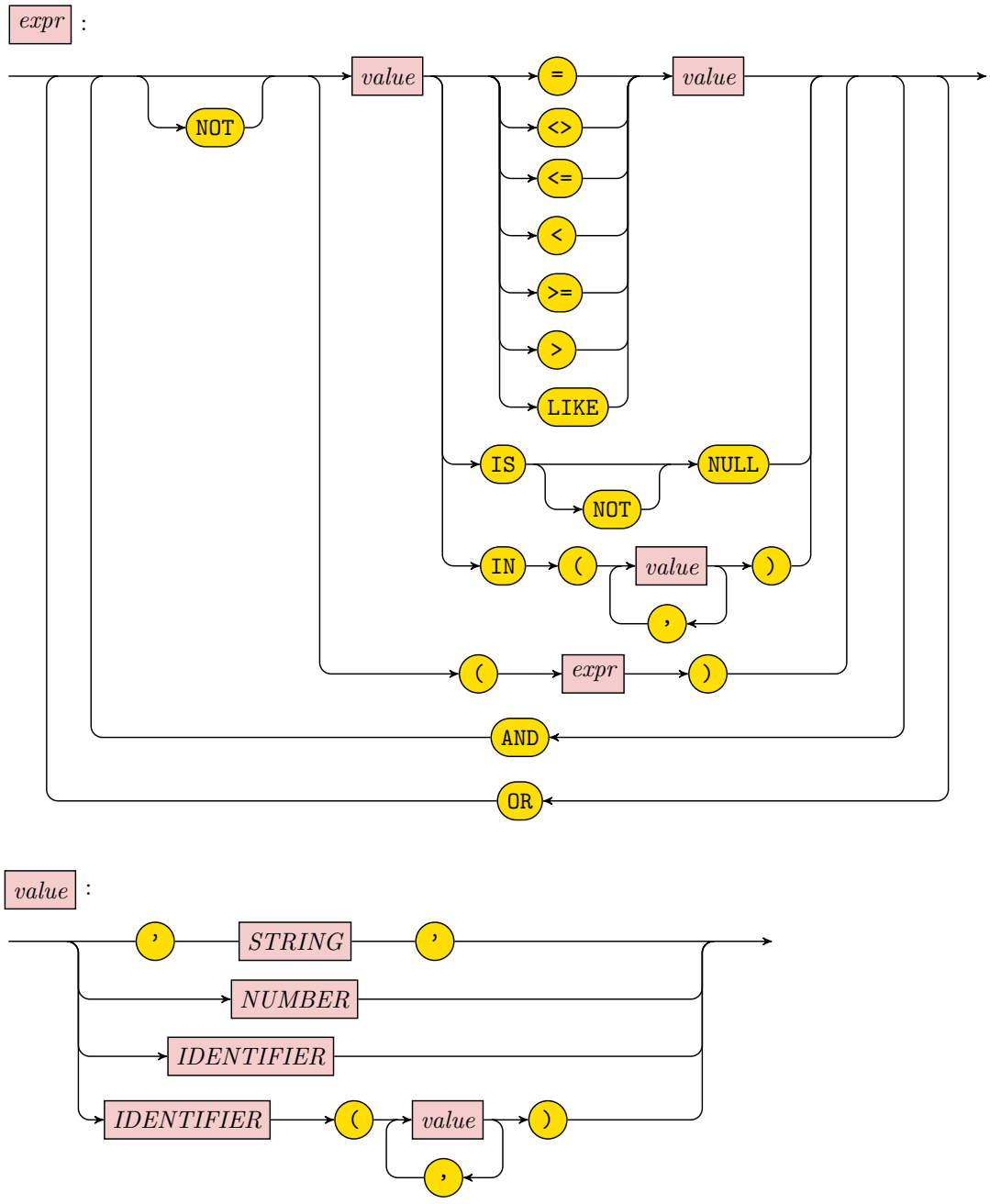


Abbildung 5.4: Syntaxdiagramm für Wertfilter nach [Pie11b]

Ein elementarer Wertfilter-Ausdruck *expr* besteht demzufolge aus der Verkettung von zwei *value*-Termen über einen Vergleichsoperator, im Fall des einstelligen Operators (`IS [NOT] NULL`) entfällt natürlich der zweite *value*-Term. Die so erzeugten Prädikate sind jeweils zu einem Wahrheitswert auswertbar und lassen sich deswegen anhand der logischen Operatoren `AND`, `OR` und `NOT` miteinander verknüpfen sowie durch Klammern verschachteln.

Ein *value*-Term kann zu einer der vier folgenden Varianten abgeleitet werden. Einerseits ist die Angabe konstanter Terme in Form von Zeichenketten (`STRING`) oder Zahlenwerten (`NUMBER`) möglich. Andererseits lassen sich variable Terme über `IDENTIFIER` bilden. Diese stehen entweder für den Namen eines aspektspezifischen Attributs oder referenzieren einen Funktionsnamen mit der zugehörigen Menge an `value`-Termen als Parameter in Klammern. Auf die dabei zu berücksichtigenden Einschränkungen für Bezeichner wird in Abschnitt 5.2.3.6 näher eingegangen.

Trotz der offensichtlichen Nähe zur WHERE-Klausel einer SQL-Anfrage stellt die Wertfilter-Syntax in Abbildung 5.4 nur einen vergleichsweise geringen Teil der direkten Vergleichsmöglichkeiten für Attributwerte dar. Darüber hinaus fehlen einige mathematischen Operatoren wie `+`, `*` und `MOD`. Da jedoch die syntaktische Erweiterung der Wertfilter-Sprache prinzipiell keine Schwierigkeit darstellt, soll im weiteren Verlauf in Abschnitt 5.1.3 vorrangig deren Wirkungsweise und Anwendbarkeit näher betrachtet werden.

Bemerkung. Trotz einer hier nicht weiter formalisierten Definition zur Semantik der Wertfilter-Sprache sollte diese durch Anlehnung an SQL intuitiv klar sein. In diesem Kontext ist zu beachten, dass die Auswertung von Wertfiltern aufgrund der in Abbildung 4.17 skizzierten Architektur nur für signierte Tupel gemäß Definition 4.4 in Abschnitt 4.2.2 semantisch sinnvoll ist.

Beispiel. Anhand der beispielhaften Einschränkung einer Anfrage sollen die grundlegenden Syntax-Elemente der Wertfilter-Sprache verdeutlicht werden.

Price $\leq$ 50 `AND` *(LCASE(Name) LIKE 'screw%'* `OR` *LCASE (Name)* `IN` *('nut', 'bolt'))*

Der so definierte Wertfilter liefert im Anfrageergebnis alle Datenzeilen, deren *Preis*-Attribut höchstens den Wert 50 hat und im Attribut *Name* entweder mit der Zeichenkette 'screw' beginnen oder den Wert 'nut' bzw. 'bolt' haben. Zur Formulierung kommen neben den logischen Operatoren `AND` und `OR` auch der Funktionsterm *LCASE* sowie die Prädikate $\leq$, *LIKE* und `IN` zum Einsatz.

5.1.3 Datentransformation

Nachdem in Abschnitt 5.1.1 die Alternativen zur Integration einer Zugriffsschicht erläutert und für diese Arbeit auf das Szenario in Abbildung 5.2 festgelegt wurden, sollen nun deren eigentliche Aufgaben im Vordergrund stehen. Hierbei spielt die Transformation aspektspezifischer Daten zwischen der signaturbasierten Darstellung für die Anwendung und den relationalen Strukturen des Referenzmodells eine zentrale Rolle, wie dies konzeptuell in Abbildung 4.12 skizziert ist. Nachfolgend werden insbesondere die Zuständigkeiten sowie die Realisierung der Filtermöglichkeiten aus Abschnitt 5.1.2 näher betrachtet.

5.1.3.1 Aufgabenteilung

Aufgrund der Tatsache, dass die Speicherung der (aspektspezifischen) Daten vollständig in der Datenbank erfolgt, wäre eine integrierte Transformation und Aufbereitung durchaus sinnvoll. Leider offenbaren diesbezügliche Tests in [Lie11b, LP10] und Kapitel 6 große Performance-Probleme für DBMS-Produkte. Auch der gegensätzliche Ansatz, in welcher die Zugriffsschicht alle Transformationen eigenständig und ohne Nutzung der relationalen Mächtigkeit des DBMS ausführt, ist mit Nachteilen hinsichtlich der Kommunikation verbunden. Entweder müssen bei jeder Anfrage große Mengen irrelevanter Daten transportiert und danach gefiltert oder sukzessive durch die Zugriffsschicht nachgeladen werden.

Dagegen erscheint der Kompromiss zwischen beiden Extrem-Konstellationen als erfolgversprechende Alternative. Datenbankseitig werden dazu die Tabellen AspectValue und AspectAssign mittels EQUI JOIN verknüpft, um die benötigten Datensätze zu ermitteln. Zusätzlich lassen sich noch die Tupel in AspectValue entsprechend filtern, da gemäß Abschnitt 4.1 aspektspezifische Daten immer einer fachlichen Tabelle t zugeordnet sind und üblicherweise nur bestimmte Attribute $c_1, \ldots, c_n$ angefragt werden. Der nachstehende Ausdruck in Relationenalgebra fasst die Aufgabe für das DBMS formal zusammen.

$$\left(\sigma_{Table=t \wedge Column \in \{c_1,\ldots,c_n\}} \text{AspectValue}\right) \underset{AspValID=AspectValue}{\bowtie} \text{AspectAssign} \quad (5.10)$$

Für die anschließende erste Pivotisierung nach den Attributwerten gemäß Schritt (i) in Abbildung 4.12 ist eine Gruppierung der Ergebnisrelation nach RowID und AspValID notwendig. Trotz ausgereifter Sortierfunktionalitäten in den DBMS-Produkten zeigte sich während der Entwicklungstests, dass die vorsortierte Rückgabe bei großen Kardinalitäten enorme Performanceverluste verursacht. Deswegen wurde diese Aufgabe abweichend vom üblichen Vorgehen der Zugriffsschicht zugeordnet, wodurch jedoch eine inkrementelle Verarbeitung nicht mehr möglich ist. Stattdessen muss das Anfrageergebnis in Ausdruck (5.10) erst vollständig vom DBMS übertragen werden und lässt sich daraufhin innerhalb der Zugriffsschicht gruppieren. Der hierfür notwendige Speicherbedarf hatte allerdings keinen signifikanten Einfluss auf die in Kapitel 6 präsentierten Testergebnisse.

5.1.3.2 Aspektfilter

Gemäß Definition 4.8 bestimmt ein Aspektfilter mit dem induzierten Kontext die gültige Menge von Kontexten aspektspezifischer Daten, die zur Erfüllung einer Anfrage führen. Dabei erfordert bereits die Ermittlung des Kontexts zu einem Datensatz der Tabelle AspectValue umfangreiche Transformationen, welche wie im vorherigen Abschnitt beschrieben primär durch die Zugriffsschicht vorgenommen werden sollen. Darüber hinaus sind aufgrund der Separierung durch das EAV-Prinzip im Allgemeinen die effektiv benötigten Daten vor einer Transformation nicht bestimmbar.

Somit müssen erst die Informationen zu allen potentiellen Aspektkontexten vom DBMS zurückgegeben werden, bevor sie in der Zugriffsschicht verarbeitet und bezüglich des induzierten Kontexts gefiltert werden können. Diese fehlende Integration von Datenzugriff und Transformation hat ihre Ursache in der nach Abbildung 5.2 vorgegebenen Systemtrennung. Um dennoch die Menge der zu transportierenden Daten möglichst gering zu halten, soll eine Methode die Anfrageparameter analysieren und daraus einschränkende Bedingungen für die SQL-Generierung ableiten. Dadurch lassen sich zumindest jene Aspektausprägungen definieren, welche bei der Datenabfrage a priori ausgeschlossen werden können.

5.1.3.3 Wertfilter

Obwohl einzelne Prädikate der in Abschnitt 5.1.2.2 eingeführten Wertfilter durch das Datenbanksystem direkt verarbeitet werden könnten, ist zur vollständigen Auswertung eines Wertfilter-Ausdrucks im Allgemeinen die Transformation der beteiligten Daten notwendig. Diese in Abbildung 4.12 als Schritt (i) benannte Pivotisierung schafft überhaupt erst die strukturelle Grundlage in Form von signierten Tupeln mit zusammengehörenden Attributen in der Ergebnisrelation, auf denen sich ein Wertfilter anwenden lässt.

Gemäß der definierten Aufgabenteilung müsste somit nach Transformation der Daten auch die Anwendung der Wertfilter innerhalb der Zugriffsschicht erfolgen. Hierfür wäre jedoch die Reimplementierung aller unterstützten Operatoren mit den jeweils relevanten Datentypen unter Berücksichtigung der spezifischen DBMS-Besonderheiten erforderlich. Eine effizientere Alternative bietet dagegen die Zerlegung des Wertfilter-Ausdrucks in seine elementaren Prädikate und deren Verarbeitung durch das DBMS. Auf Basis dieser Ergebnisse kann die Zugriffsschicht zunächst die erwähnte Datentransformation durchführen und anschließend durch Iteration über die logischen Operatoren zwischen den Prädikaten (siehe Abbildung 5.4) die Gesamtauswertung des Wertfilter-Ausdrucks gewährleisten.

Voraussetzung für diesen Ansatz ist jedoch die Beschränkung auf Prädikate, die nur genau ein aspektspezifisches Attribut referenzieren. Demzufolge wäre beispielsweise das Prädikat "*Name = Description*" nicht zulässig, das Beispiel aus Abschnitt 5.1.2.2 hingegen schon. Dies ergibt sich aus der Tatsache, dass im DBMS für die Prädikat-Auswertung nur auf Tupel aus der Tabelle AspectValue zugegriffen werden kann, welche jeweils Informationen zu einem Attributwert beinhalten. Die Transformation zusammengehörender Attribute und Werte zu signierten Tupeln findet wie erläutert erst in der Zugriffsschicht statt.

5.1.3.4 Fazit

Basierend auf den vorangegangenen Betrachtungen soll nun der generelle Ablauf beim Umgang mit aspektspezifischen Daten unter Nutzung der Zugriffsschicht zusammenfassend dargestellt werden. Trotz der Fokussierung auf einen lesenden Zugriff lässt sich dieses Vorgehen analog auf modifizierende Anwendungsfälle übertragen. Eine dabei gegebenenfalls erforderliche Verknüpfung mit aspektunabhängigen Daten in den fachlichen Basistabellen muss gemäß Abbildung 4.17 in der Anwendung erfolgen.

Zur Initialisierung erzeugt die Anwendung eine entsprechend parametrisierte Anfrage-Instanz über die API der Zugriffsschicht, welche dadurch für die weitere Abarbeitung zuständig ist. Im nächsten Schritt werden nun alle übergebenen Parameter ausgewertet, was bei Festlegung eines Wertfilters die Extraktion der einzelnen Prädikate beinhaltet. Darüber hinaus findet eine Prüfung statt, inwieweit ein spezifizierter Aspektfilter eine Reduzierung der betrachteten Daten von vornherein ermöglicht. Die Gesamtheit jener gewonnenen Informationen beeinflusst schließlich die SQL-Anfrage, welche generiert und an das DBMS geschickt wird.

Innerhalb des DBMS wird gemäß Ausdruck (5.10) nur der Verbund zwischen AspectValue und AspectAssign vollzogen. In diesem Zuge werden gegebenenfalls Wertfilter-Prädikate sowie generelle Einschränkungen eines Aspektfilters realisiert. Die Ergebnisrelation steht daraufhin zur weiteren Verarbeitung an der Zugriffsschicht bereit. Hier findet zunächst die erste Pivotisierung nach den Attributwerten statt, die eine nach RowID sortierte Tabelle mit allen Aspektzuordnungen zu jedem aspektspezifischen Wert erzeugt. Im

Rahmen der zweiten Pivotisierung erfolgt die Zusammenführung von Aspektwerten mit identischen Signaturen zu uniformen Tupeln. Auf diesen können nun vorgegebene Wert- und Aspektfilter angewendet werden. Die somit erzeugten Ausgabedaten stehen als Rückgabe der Anfrageinstanz der Anwendung zur Verfügung. Eine schematische Darstellung des gesamten Ablaufs ist in **Abbildung 5.5** gegeben.

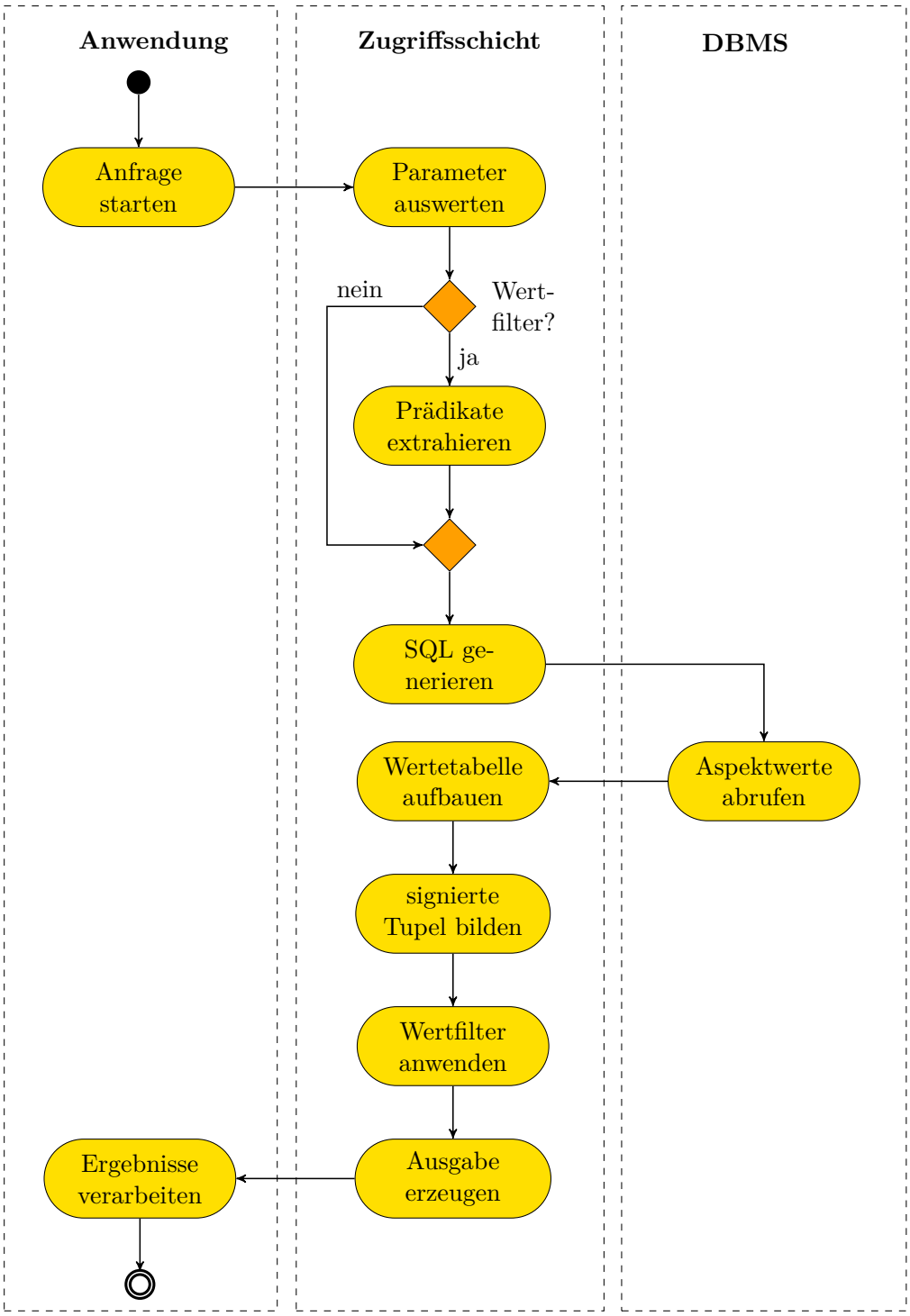


Abbildung 5.5: UML-Aktivitätsdiagramm zur Anfrageverarbeitung nach [Pie11b]

5.2 Beschreibung der API

Zur Gewährleistung des in Abbildung 5.5 skizzierten Ablaufs für die Verarbeitung aspektspezifischer Daten muss die Zugriffsschicht einen entsprechenden Funktionsumfang aufweisen. Dieser Abschnitt spezifiziert mittels Java und der Unified Modeling Language (UML) Schnittstellen-Definitionen und Klassen im eigentlichen Sinn einer API, welche den Zugriff auf die Strukturen und Konzepte des Referenzmodells aus Kapitel 4 ermöglichen. Dabei beschränkt sich die Beschreibung auf Methodensignaturen, zusätzliche Informationen wie Ausnahmebehandlungen werden aus Gründen der Übersicht ausgeblendet. Ebenso erfolgt keine formale Spezifikation der funktionalen Anforderungen an die API, stattdessen werden an geeigneter Stelle wichtige Details einer Referenzimplementierung vorgestellt. Die beispielhafte Anwendung der so definierten Funktionalität der Zugriffsschicht wird später im Abschnitt 5.3 demonstriert.

Die Visualisierung jener statischen Implementierungs-Aspekte erfolgt sowohl in Form von UML-Klassendiagrammen gemäß [UML10] als auch durch Einbettung von Listings mit Java-Code. Die gesamte Erläuterung der API ist zudem in thematische Einheiten gegliedert, welche darüber hinaus teilweise den Entwicklungsprozess der Zugriffsschicht widerspiegeln. Zu Beginn wird in Abschnitt 5.2.1 der Aspektmanager als zentraler Einstiegspunkt für die Zugriffsschicht vorgestellt. Da eine Nutzung der API ohne Metadaten zum Aspektkatalog nicht sinnvoll möglich ist, werden daraufhin entsprechende Methoden in Abschnitt 5.2.2 eingeführt. Anschließend steht die Spezifikation von Filterbedingungen in Abschnitt 5.2.3 im Vordergrund, bevor der Zugriff auf aspektspezifische Daten in Abschnitt 5.2.4 betrachtet wird. Schließlich finden sich in Abschnitt 5.2.5 einige Implementierungsdetails hinsichtlich der generellen Anfrage-Verarbeitung.

5.2.1 Aspektmanager

Die Verwendung der Zugriffsschicht erfolgt durch Aufruf zentraler Methoden im Interface **AspectManager**, welche in **Listing 5.1** dargestellt sind und von der Referenzimplementierung in einer entsprechenden Klasse realisiert werden. Zusätzliche typische Aufgaben wie beispielsweise der Aufbau einer Datenbank-Verbindung sind dagegen von der konkreten Anwendung abhängig und daher nicht Teil dieses abstrahierenden Interfaces.

```
interface AspectManager {
    public AspectCatalogManager getAspectCatalogManager ();
    public AspectContext createAspectContext ();
    public AspectContext createAspectContext (AspectSet set);
    public UnambiguousAspectContext createUnambiguousAspectContext ();
    public UnambiguousAspectContext createUnambiguousAspectContext (AspectSet set);
    public AspectSet createAspectSet (int[] aspects);
    public AspectFilter createAspectFilter (AspectFilter.FilterNode node);
    public AspectFilter createAspectFilter (String infixAspectFilter);
    public ModifyStatement createModifyStatement (int tabID);
    public QueryStatement createQueryStatement (int tabID);
    public ColumnSet createColumnSet (int[] columns);
}
```

Listing 5.1: Interface AspectManager

Mit Hilfe der Methode *getAspectCatalogManager* kann eine Instanz des Aspektkatalog-Managers erzeugt werden, der für den Zugriff auf die Metadaten des Aspektkatalogs zuständig ist. Die genaue Beschreibung jener Schnittstelle erfolgt im nachfolgenden Abschnitt 5.2.2.

Dagegen stellt **AspectManager** unterschiedliche Factory-Methoden zur Verfügung, um Datenstrukturen und Filterkriterien zu spezifizieren. Dies betrifft einerseits die in Abschnitt 4.2.3 eingeführten (eindeutigen) Aspektkontexte, welche sich mit den Methoden *createAspectContext* bzw. *createUnambiguousAspectContext* erzeugen lassen. Dabei werden bei der jeweils parameterlosen Variante alle im Aspektkatalog registrierten Aspekte berücksichtigt, während sich diese Menge durch Übergabe eines **AspectSet** auch einschränken lässt. Dessen Instanziierung erfolgt durch Übergabe eines Arrays mit den Aspekt-IDs an die Methode *createAspectSet*. Andererseits können auch Ausprägungen von **AspectFilter** erzeugt werden, welche je nach Parametrisierung der dafür zuständigen Methode *createAspectFilter* die Aspektfilter aus Abschnitt 4.2.4 bzw. die Infix-Aspektfilter gemäß Abschnitt 5.1.2.1 repräsentieren.

Darüber hinaus ermöglichen die Methoden *createModifyStatement* und *createQueryStatement* die Instanziierung von Klassen für den Zugriff auf aspektspezifische Daten. Hierbei ist jeweils die ID derjenigen fachlichen Tabelle anzugeben, auf die sich eine (ändernde) Anfrage beziehen soll. Für die Festlegung der dabei relevanten Tabellenspalten kann mittels *createColumnSet* ein entsprechendes **ColumnSet**-Objekt angelegt werden. Details zur Verwendung und den Schnittstellen der in diesem Kontext erstellten Objekte finden sich in Abschnitt 5.2.4.

5.2.2 Aspektkatalog-Anfragen

Basierend auf den in Abbildung 4.3 präsentierten Tabellen des Referenzmodells erfolgt die Identifizierung aller Metadaten über ganzzahlige Idents, ausgenommen von diesem Prinzip sind die Aspektabhängigkeiten. Gemäß der Argumentation in Abschnitt 3.1 sind funktionale Aspekte und damit deren Metadaten in einem Datenmodell jedoch nicht statisch definierbar, sondern unterliegen den sich ständig ändernden fachlichen Anforderungen. Aus diesem Grund muss die API entsprechende Möglichkeiten bereitstellen, um die Daten des Aspektkatalogs auslesen und manipulieren zu können. Für diese Aufgaben kommt das in **Listing 5.2** dargestellte Interface **AspectCatalogManager** zum Einsatz, dessen Methoden in den nachfolgenden Abschnitten näher erläutert werden.

Die Referenzimplementierung unterliegt dennoch der praxisnahen Prämisse, dass sich die Aspekt-Metadaten gegenüber den fachlichen Daten eher selten ändern. Eine weitere Voraussetzung ist die Zwischenspeicherung jener Katalogdaten[3] im Arbeitsspeicher, was aufgrund des eher geringen Datenvolumens problemlos realisierbar sein sollte. Somit ist es möglich, alle Änderungen am Aspektkatalog innerhalb einer Anwendung und dem Datenbanksystem konsistent zu halten. Resultieren allerdings derartige Modifikationen von weiteren API-Instanzen oder aus Ad-hoc-Zugriffen ohne Nutzung der Methoden in Listing 5.2, bleiben sie von anderen Anwendungen unbemerkt. Eine Lösung jener Problematik wird hier bewusst nicht weiter betrachtet, die Diskussion potentieller Strategien und sinnvoller Erweiterungen findet in Kapitel 7 statt.

[3] Im Gegensatz dazu wird gemäß Abbildung 5.2 auf das Caching der aspektspezifischen Daten aus den Tabellen AspectValue und AspectAssign verzichtet.

```
interface AspectCatalogManager {
    /** datatype methods **/
    public int[] getAllDatatypeIDs ();
    public String getDatatypeBaseName (int typeID);
    public int getDatatypeLength (int typeID);
    public int getDatatypeScale (int typeID);
    public int addDatatype (String typeName, int length);
    public int addDatatype (String typeName, int length, int scale);
    public void deleteDatatype (int typeID);

    /** aspect methods **/
    public int[] getAllAspectIDs ();
    public int lookupAspectID (String aspectName);
    public String lookupAspectName (int aspID);
    public String lookupAspectKey (int aspID);
    public int lookupAspectDatatypeID (int aspID);
    public int addAspect (String aspectName, String aspectKey, int typeID);
    public void deleteAspect (int aspID);
    public int[] getAspectKeyValueIDs (int aspID);
    public int lookupAspectKeyValueID (int aspID, String keyValueName);
    public String lookupAspectKeyValueName (int keyValueID);
    public int addAspectKeyValue (int aspID, String keyValue, String comment);
    public void deleteAspectKeyValue (int keyValueID);

    /** base table meta data **/
    public int[] getAllTableIDs ();
    public int lookupTableID (String schemaName, String tableName);
    public String lookupTableName (int tabID);
    public String lookupTableSchema (int tabID);
    public String lookupTableKeyName (int tabID);
    public int addTable (String schemaName, String tableName, String keyName);
    public void deleteTable (int tabID);
    public int[] getColumnIDs (int tabID);
    public int lookupColumnID (int tabID, String columnName);
    public String lookupColumnName (int colID);
    public int lookupColumnDatatypeID (int colID);
    public int addColumn (int tabID, String columnName, int typeID);
    public void deleteColumn (int colID);

    /** dependencies **/
    public boolean depends (int colID, int aspID);
    public int[] getTableDependencies (int tabID);
    public int[] getColumnDependencies (int colID);
    public int[] getColumnDependencies (int[] colIDs);
    public int[] getColumnDependencies (ColumnSet colSet);
    public int[] getAspectDependentTables (int aspID);
    public int[] getAspectDependentColumns (int aspID);
    public void addDependency (int colID, int aspID);
    public void deleteDependency (int colID, int aspID);
}
```

Listing 5.2: Interface AspectCatalogManager

Zur Gewährleistung der Zwischenspeicherung umfasst die Referenzimplementierung von **AspectCatalogManager** jeweils eine eigene Klasse für die Tabellen AspectTable, AspectColumn, AspectDatatype, AspectDefinition sowie AspectKeyValue. Deren innere Struktur ohne Angabe der Konstruktoren ist als UML-Klassendiagramm in **Abbildung 5.6** dargestellt. Hierbei wird die große Übereinstimmung mit Abbildung 4.3 deutlich sichtbar, welche aus dem verwendeten objektrelationalen Mapping resultiert. Nur die Transformation der Aspektabhängigkeiten in Tabelle AspectDependence wurde nicht über dieses Konzept umgesetzt. Stattdessen kommen zwei einfache Hashtabellen zum Einsatz, um für eine Column-ID die Menge der abhängigen Aspekt-IDs sowie für eine Aspekt-ID die Menge der beeinflussten Column-IDs hinterlegen zu können.

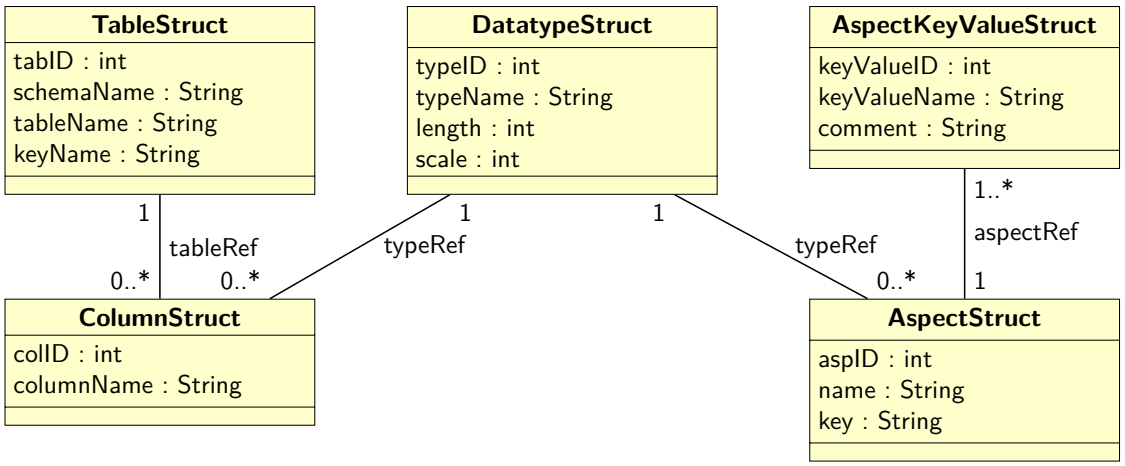


Abbildung 5.6: UML-Klassendiagramm zum Aspektkatalog-Mapping nach [Pie11b]

5.2.2.1 Metadaten zu Datentypen

Durch Aufruf der Methode *getAllDatatypeIDs* liefert die Zugriffsschicht eine Liste mit Idents für alle im Aspektkatalog registrierten Datentypen. Zu einer Datentyp-ID können daraufhin mittels *getDatatypeBaseName*, *getDatatypeLength* und *getDatatypeScale* jeweils der zugehörige Basistyp[4], die Länge und die Genauigkeit abgefragt werden. Für die Definition eines neuen Datentyps ist die Methode *addDatatype* zu verwenden, bestehende Einträge in der Tabelle AspectDatatype können mit *deleteDatatype* gelöscht werden.

5.2.2.2 Metadaten zu Aspekten

Durch Aufruf der Methode *getAllAspectIDs* liefert die Zugriffsschicht eine Liste mit Idents für alle im Aspektkatalog registrierten funktionalen Aspekte. Alternativ lässt sich eine einzelne ID auch über die Methode *lookupAspectID* ermitteln, wenn der Name eines Aspekts (beispielsweise „Sprache“) bekannt ist. Zu einer solchen Aspekt-ID können daraufhin mittels *lookupAspectName*, *lookupAspectKey* und *lookupAspectDatatypeID* der Name, der Bezeichner des Aspektschlüssels und die Datentyp-ID abgefragt werden. Für die Definition eines neuen Aspekts dient die Methode *addAspect*, bestehende Einträge in der Tabelle AspectDefinition können mit *deleteAspect* gelöscht werden.

[4]Bezeichnungen sind beliebig wählbar ohne Beschränkung auf vordefinierte Typen der SQL-Norm.

Darüber hinaus lassen sich die Idents aller Schlüsselwerte zu einem Aspekt durch Aufruf der Methode *getAspectKeyValueIDs* ermitteln. Alternativ kann auch eine einzelne ID durch *lookupAspectKeyValueID* auf Basis der übergebenen Aspekt-ID und dem betreffenden Schlüsselwert bestimmt werden. Umgekehrt liefert *lookupAspectKeyValueName* den Schlüsselwert zu einer Aspektschlüsselwert-ID. Für die Definition eines neuen Aspektschlüsselwerts dient die Methode *addAspectKeyValue*, bestehende Einträge in der Tabelle AspectKeyValue können mit *deleteAspectKeyValue* gelöscht werden.

5.2.2.3 Metadaten zu Basistabellen

Durch Aufruf der Methode *getAllTableIDs* liefert die Zugriffsschicht eine Liste mit Idents für alle im Aspektkatalog registrierten fachlichen Basistabellen. Alternativ lässt sich eine einzelne ID auch über die Methode *lookupTableID* ermitteln, wenn Name und Schema der Tabelle bekannt sind. Zu einer solchen Tabellen-ID können daraufhin über *lookupTableName*, *lookupTableSchema* und *lookupTableKeyName* der Tabellenname, das Schema und der Name des ein-elementigen Schlüssels abgefragt werden. Dieses Schlüsselattribut ist gemäß Abschnitt 4.3.2 für die Identifizierung eines Datensatzes der Basistabelle zuständig, wobei die RowID in AspectValue entsprechende Werte referenziert. Für die Registrierung einer weiteren Basistabelle dient die Methode *addTable*, bestehende Einträge in der Katalogtabelle AspectTable können mit *deleteTable* gelöscht werden.

Darüber hinaus lassen sich die Idents aller Spalten zu einer Basistabelle durch Aufruf der Methode *getColumnIDs* abfragen. Alternativ können über *lookupColumnID* einzelne Spalten-IDs zu einer Tabelle und einem Spaltennamen bestimmt werden. Umgekehrt liefern *lookupColumnName* und *lookupColumnDatatypeID* die Bezeichnung sowie den Datentyp zu einer tabellenübergreifend eindeutigen Spalten-ID. Für die Registrierung neuer Spalten zu einer im Aspektkatalog bekannten Basistabelle dient die Methode *addColumn*, bestehende Einträge in der Tabelle AspectColumn können mit *deleteColumn* wieder gelöscht werden.

5.2.2.4 Metadaten zu Aspektabhängigkeiten

Die Zugriffsschicht stellt mit der Methode *depends* eine Möglichkeit zur Verfügung, zu prüfen, ob die Spalte einer Basistabelle von einem ganz konkreten funktionalen Aspekt abhängt. Dagegen kann die Menge aller auf eine Basistabelle wirkenden Aspekte mit der Methode *getTableDependencies* ermittelt werden. Analog liefert *getColumnDependencies* jeweils jene Menge von Aspekt-IDs, die auf eine einzelne Spalte oder eine Menge von Spalten wirken. Dabei lässt sich die Spaltenmenge entweder als ID-Liste oder als **ColumnSet**-Objekt angeben, nähere Erläuterungen zu dieser Klasse finden sich in Abschnitt 5.2.4. Umgekehrt existieren mit *getAspectDependentTables* und *getAspectDependentColumns* zwei Methoden, um für einen funktionalen Aspekt die Menge der beeinflussten Basistabellen bzw. Spalten zu bestimmen. Mit Hilfe von *addDependency* lässt sich eine neue Abhängigkeit zwischen einer Spalte und einem Aspekt hinzufügen, bestehende Einträge in der Tabelle AspectDependence können mittels *deleteDependency* gelöscht werden.

5.2.3 Filterbedingungen

Nachdem im vorangegangenen Abschnitt 5.2.2 Methoden für den Zugriff auf die Katalogtabellen des Referenzmodells betrachtet wurden, steht nun die Realisierung der Begriffe und theoretischen Grundlagen gemäß Abschnitt 4.2 im Vordergrund. Hierzu werden geeignete Datentypen und Klassen zur Spezifikation von Filterbedingungen vorgestellt.

5.2.3.1 Aspektmengen

Bei der Arbeit mit aspektspezifischen Daten ist sowohl für die Formulierung von Filterbedingungen als auch für die Auswertung von Anfrageergebnissen die Menge der jeweils relevanten Aspekte notwendig. Dabei spielt natürlich deren Reihenfolge im Sinne der Aspektnummerierung aus Definition 4.3 eine entscheidende Rolle. Von dieser abzählbaren Aspektmenge werden typischerweise mehrere verschiedene Teilmengen benötigt aufgrund der vielfältigen Aspektabhängigkeiten in den fachlichen Basistabellen. Das zugehörige Interface **AspectSet** ist in **Listing 5.3** dargestellt.

```
interface AspectSet {
    public int count ();
    public int getIndex (int aspID);
    public int getAspect (int index);
    public int[] toArray ();
}
```

Listing 5.3: Interface AspectSet

Eine Instanz von **AspectSet** lässt sich mittels *createAspectSet* am **AspectManager** durch Übergabe einer Menge von Aspekt-IDs erzeugen. Die dabei erfolgte Durchnummerierung der Aspekte bildet die Grundlage für die Methoden *getIndex* und *getAspect*, welche zu einer Aspekt-ID bzw. einem Index die jeweils zugehörige Information liefern. Zur Bestimmung der Kardinalität der Aspektmenge ist die Methode *count* aufzurufen, die ursprüngliche Liste der Aspekt-IDs wird von *toArray* zurückgegeben.

5.2.3.2 Aspektsignaturen

Für die Festlegung aspektspezifischer Daten sind neben der zugehörigen Aspektmenge insbesondere deren jeweilige Schlüsselwerte von Bedeutung. Diese Zuordnung wird gemäß Abschnitt 4.2.2 mit Hilfe der Aspektsignaturen definiert. Die wichtigsten Methodenrümpfe der Klasse **AspectSignature** sind in **Listing 5.4** aufgeführt.

```
private class AspectSignature {
    private AspectSignature (AspectSet set) {}
    private void setIndexKeyValue (int aspIndex, int keyID) {}
    private int getIndexKeyValue (int aspIndex) {}
    private boolean equals (Object obj) {}
    private int hashCode () {}
}
```

Listing 5.4: Klasse AspectSignature (unvollständig)

Bei der Instanziierung erwartet der Konstruktor ein **AspectSet** als Parameter und legt zu diesem ein Array der Länge *AspectSet.count* für die Speicherung der Aspektschlüsselwerte an. Diese können anschließend mit *setIndexKeyValue* zugeordnet und mit *getIndexKeyValue* gelesen werden. Weiterhin findet für die beiden von **java.lang.Object** vererbten Methoden *equals* und *hashCode* eine Überschreibung statt, um die Menge der enthaltenen Aspekte sowie ihrer Aspektschlüsselwerte inhaltlich auswerten zu können. Dies ist notwendig, da standardmäßig diese Methoden für den Vergleich bzw. den Hashcode die Objekt-Speicheradressen nutzen und damit eine Verwendung von Aspektsignaturen als Schlüssel für Speicher-Hashtabellen nicht sinnvoll wäre.

Die Verwendung von Aspektsignaturen beschränkt sich jedoch auf interne performancekritische Transformationsprozesse zwischen der Persistenz- und Zugriffsschicht, die in Abbildung 4.12 schematisch dargestellt sind. Dementsprechend ist **AspectSignature** nicht als öffentliche Klasse verfügbar und unterliegt bei der Implementierung im besonderen Maße der Berücksichtigung von Laufzeitaspekten. Stattdessen erfolgt seitens der Anwendung die Spezifikation und Verarbeitung von Aspekten für aspektspezifische Daten mit Hilfe der im nachfolgenden Abschnitt vorgestellten Aspektkontexte.

5.2.3.3 Aspektkontexte

Während Aspektsignaturen ausschließlich auf der Persistierungsebene eine Rolle spielen, dienen Aspektkontexte zur Einschränkung der für eine Anfrage relevanten signierten Tupel. Ein Aspektkontext selbst besteht entsprechend der Definition in Abschnitt 4.2.3 aus einer Menge von Tupeln mit Aspektschlüsselwerten. Damit ist ein konkreter Datensatz für eine Anfrage in der zu betrachtenden Datenmenge enthalten, wenn seine Signatur mit einem solchen Aspektschlüssel-Tupel übereinstimmt. Für deren Abbildung stellt die API das in **Listing 5.5** gezeigte Interface **AspectContextElement** bereit.

```
interface AspectContextElement {
    public static final int VALUE_NULL = −1;
    public AspectContext getAspectContext ();
    public void setIndexKeyValue (int aspIndex, int keyValueID);
    public void setIndexKeyNull (int aspIndex);
    public int getIndexKeyValue (int aspIndex);
    public boolean isIndexKeyNull (int aspIndex);
}
```

Listing 5.5: Interface AspectContextElement

Zu jedem Aspektkontextelement lässt sich über die Methode *getContext* der zugehörige Aspektkontext ermitteln, dessen Klassenstruktur nachfolgend näher betrachtet wird. Mit Hilfe der Methode *setIndexKeyValue* kann für einen Aspekt (aspIndex) sowohl ein konkreter Aspektschlüsselwert (keyValueID) als auch kein Schlüsselwert im Sinne einer Belegung mit $\perp$ gemäß Definition 4.1 festgelegt werden. Umgekehrt gibt *getIndexKeyValue* den aktuellen Schlüsselwert zum referenzierten Aspekt zurück. Wenn dieser unbelegt ist, liefert die Methode *isIndexKeyNull* eine wahre Aussage. Dabei gilt für die ID des jeweils über aspIndex angesprochenen Aspekts:

$$\mathsf{ID} = \mathit{getContext}().\mathit{getAspectSet}().\mathit{getAspectID}(\mathsf{aspIndex})$$

Die Zusammenfassung mehrerer derartiger Kontextelemente zu einem Aspektkontext ist über das in **Listing 5.6** dargestellte Interface **AspectContext** möglich. Dessen Instanziierung erfolgt wie in Abschnitt 5.2.1 vorgestellt über Methoden von **AspectManager**.

```
interface AspectContext {
    public AspectSet getAspectSet ();
    public int elementCount ();
    public AspectContextElement getElement (int index);
    public AspectContextElement addElement ();
}
```

Listing 5.6: Interface AspectContext

Durch Aufruf der Methode *getAspectSet* kann die Menge der für einen Kontext relevanten Aspekte ermittelt werden, wobei die resultierende Struktur **AspectSet** implizit eine Ordnung jener Aspekte beinhaltet. Änderungen an der im Katalog registrierten Gesamtmenge von Aspekten über entsprechende in Abschnitt 5.2.2.2 beschriebene Methoden führen somit zur Ungültigkeit von Aspektkontexten. Darüber hinaus lässt sich mit der Methode *elementCount* die Anzahl der zu einem Aspektkontext bereits erfassten Aspektkontextelemente abfragen. Der Zugriff auf ein solches Element wird über die Methode *getElement* gewährleistet, eine Erweiterung dieser Menge ist mittels *addElement* möglich.

Für den Spezialfall, dass ein Aspektkontext nur ein Kontextelement enthält, stellt die API das von **AspectContext** abgeleitete Interface **UnambiguousAspectContext** zur Verfügung, welches eine Initialisierung als Kernaspektkontext nach Definition 4.6 voraussetzt. Wie in **Listing 5.7** gezeigt, wird die Methode *getElement* um eine parameterlose Variante erweitert, welche den Zugriff auf das eine Kontextelement bietet. Die Implementierung der Methode *addElement* muss dagegen mit einer Ausnahme abbrechen.

```
interface UnambiguousAspectContext extends AspectContext {
    public AspectContextElement getElement ();
}
```

Listing 5.7: Interface UnambiguousAspectContext

5.2.3.4 Aspektfilter

Wie in Abschnitt 4.2.4 motiviert, ist die Spezifikation von Aspekt-Einschränkungen mit Aspektfiltern gegenüber Aspektkontexten sowohl intuitiver als auch kompakter aufgrund der deklarativen Sprache in Definition 4.7. Derartige Aspektfilter lassen sich als Baumstruktur interpretieren, wobei Operatoren und elementare Ausdrücke auf den Blättern jeweils als Knoten durch das Interface **FilterNode** in **Listing 5.8** repräsentiert werden.

```
interface FilterNode {
    public int matches (AspectContextElement contextElement);
    public String toInfixString ();
}
```

Listing 5.8: Interface FilterNode

Durch Aufruf der Methode *matches* kann geprüft werden, ob ein solcher Knoten das übergebene Aspektkontextelement im Sinne seiner charakteristischen Funktion gemäß Definition 4.8 akzeptiert. Zudem liefert die Methode *toInfixString* die Darstellung eines Filterausdrucks in Infix-Schreibweise, welche in Abschnitt 5.2.3.5 näher betrachtet wird. Basierend auf **FilterNode** lässt sich schließlich ein Aspektfilter von der Baumwurzel beginnend aufbauen. Hierfür notwendige Methoden definiert das in **Listing 5.9** dargestellte Interface **AspectFilter**. Dessen Instanziierung erfolgt mit Hilfe der Methode *createAspectFilter* in **AspectManager** und einer **FilterNode**-Instanz als Parameter.

```
interface AspectFilter {
    public static interface FilterNode;

    public static FilterNode keyValue (int aspID, int aspKey);
    public static FilterNode keyNull (int aspID);
    public static FilterNode and (FilterNode op1, FilterNode op2);
    public static FilterNode and (FilterNode []operands);
    public static FilterNode or (FilterNode op1, FilterNode op2);
    public static FilterNode or (FilterNode []operands);
    public static FilterNode not (FilterNode operand);

    public void setFilter (FilterNode node);
    public FilterNode getRootFilterNode ();
    public boolean matches (AspectContext context);
    public String toInfixString ();
}
```

Listing 5.9: Interface AspectFilter

Analog zum induktiven Aufbau in Definition 4.7 dienen die Methoden *keyValue* und *keyNull* der Erzeugung elementarer Filterausdrücke als **FilterNode**-Objekte. Dem durch aspID referenzierten Aspekt wird dabei ein Schlüsselwert bzw. eine Belegung mit dem Nullwert $\perp$ zugewiesen. Darauf aufbauend lassen sich zusammengesetzte Filterausdrücke mit den Methoden *and*, *or* sowie *not* erstellen, wobei **FilterNode**-Instanzen als Parameter erwartet werden. Zu den mehrstelligen Operatoren *and* und *or* existieren jeweils zwei Methoden. Neben der allgemeinen Variante zur Verknüpfung beliebig vieler Unterausdrücke eines **FilterNode**-Arrays wird als Vereinfachung auch eine Form mit genau zwei Operanden angeboten, welche zudem in jenen Fällen die Code-Lesbarkeit erhöht.

Weiterhin kann an einem **AspectFilter** der zugehörige Filterausdruck durch die Methode *setFilter* ersetzt werden, *getRootFilterNode* gibt den äußersten Filterausdruck im Sinne des obersten Wurzelelements in einer Baumstruktur zurück. Zur Überprüfung der Akzeptanz eines Aspektkontexts durch einen konkreten Aspektfilter existiert die Methode *matches*, deren Ergebnis auf den rekursiven Aufrufen der gleichnamigen Funktion an allen **FilterNode**-Instanzen basiert. Dieses Prinzip nutzt analog auch die Methode *toInfixString*, um die Infix-Schreibweise für einen kompletten Aspektfilter zu generieren.

5.2.3.5 Infix-Aspektfilter

Alternativ zur Verwendung des Interface **AspectFilter** kann für die Spezifikation von Aspekt-Einschränkungen auch auf die intuitiveren Infix-Aspektfilter (IAF) zurückgegriffen werden. Dieses in Abschnitt 5.1.2.1 vorgestellte Konstrukt ermöglicht die Angabe einer

Zeichenkette, welche sich semantisch auf korrelierende Aspektfilter-Ausdrücke abbilden lässt. Die hierfür notwendige Vorverarbeitung durch die API ist jedoch erst während der Ausführungszeit möglich, wogegen die Syntaxprüfung von **FilterNode**-Instanzen bereits direkt durch den Java-Compiler stattfindet.

Grundlage für die Struktur und Interpretation von IAF-Zeichenketten ist das in Abbildung 5.3 definierte Syntaxdiagramm. Aus diesem ist prinzipiell eine *LL-Grammatik* ableitbar, welche mit der Technik des *Top-Down-Parsing* [Knu71] verarbeitet werden kann. Die Bereitstellung jenes Parsers erfolgt im Rahmen der Referenz-Implementierung durch Einsatz des Parser-Generators ANTLR3, der in [PQ95] näher beschrieben ist. Die hierfür notwendige (vereinfachte) Grammatik zeigt **Listing 5.10**, womit die algorithmische Transformation von IAF in Aspektfilter-Ausdrücke gemäß Definition 5.1 gegeben ist.

```
grammar IAF;

/* Parser rules */
start : EOF | expr EOF;

expr: conj ('OR' conj)*;
conj: atom ('AND' atom)*;

atom: assign
    | 'NOT' '(' expr ')'
    | 'NOT' assign
    | '(' expr ')';

assign: key (('=' value)
    | ('IS' ('NOT')? 'NULL')
    | ('IN' '(' value (',' value)* ')'));

key:   (QUOTED_CHARS)
    | ('ASPECT' NUMBER);

value: (QUOTED_CHARS)
    | (NUMBER);

/* Lexer rules */
NUMBER:          (DIGIT)+;
QUOTED_CHARS:    '\'' ( '\'\'' | ~('\'') )* '\'';
WHITESPACE:      ('\t' | ' ' | '\r' | '\n' | '\u000C')+ {$channel = HIDDEN;};
fragment DIGIT:  '0' .. '9';
```

Listing 5.10: ANTLR3-Grammatik für Infix-Aspektfilter (vereinfacht)

Tatsächlich enthält die vollständige ANTLR3-Grammatik für die Verarbeitung von IAF-Ausdrücken unter anderem die Spezifikation notwendiger Methoden zur Umwandlung geparster Teil-Ausdrücke in **FilterNode**-Objekte. Aus Gründen der Übersicht sei hier jedoch für weiterführende Details auf **Listing B.1** im Anhang verwiesen. ANTLR3 erzeugt schließlich aus jener Grammatik-Datei entsprechenden Java-Quellcode, der eine Validierung von IAF-Ausdrücken auf Basis abstrakter Syntaxbäume, englisch Abstract Syntax Tree (AST), vornimmt. Dabei stellt die Methode *createIAFParser* den Einstiegspunkt dar, welcher in der Referenz-Implementierung von **AspectManager** durch *createAspectFilter* bei Übergabe einer IAF-Zeichenkette gerufen wird.

5.2.3.6 Wertfilter

Auch die in Abschnitt 5.1.2.2 definierten Wertfilter sollen analog zu den Infix-Aspektfiltern mittels einer Zeichenkette in der Anfrage spezifizierbar sein. Aus diesem Grund wurde ausgehend vom zugehörigen Syntaxdiagramm in Abbildung 5.4 eine für ANTLR3 prozessierbare Grammatik definiert. **Listing 5.11** stellt diesbezüglich die wichtigsten Elemente dar, während die vollständige Beschreibung in **Listing B.2** im Anhang zu finden ist.

```
grammar VF;

/* Parser rules */
start : EOF | expr EOF;

expr: conj ('OR' conj)*;
conj: atom ('AND' atom)*;

atom: comp_pred
    | 'NOT' '(' expr ')'
    | 'NOT' comp_pred
    | '(' expr ')';

comp_pred:
    value ((('=' | '<>' | '<' | '>' | '<=' | '>=' | 'LIKE') rvalue)
    | ('IS' ('NOT')? 'NULL')
    | ('IN' '(' rvalue (',' rvalue)* ')'));

value:  NUMBER
    | QUOTED_CHARS
    | func '(' value (',' value)* ')'
    | IDENTIFIER;

rvalue: NUMBER
    | QUOTED_CHARS
    | func '(' rvalue (',' rvalue)* ')';

func: IDENTIFIER;

/* Lexer rules */
NUMBER:          (DIGIT)+ ('.' (DIGIT)+)? (('e'|'E') (DIGIT)+)?;
IDENTIFIER:      LETTER (LETTER | DIGIT | '_')+;
QUOTED_CHARS:    '\'' ( '\'\'' | ~('\'') )* '\'';
WHITESPACE:      ('\t' | ' ' | '\r' | '\n' | '\u000C')+ {$channel = HIDDEN;};
fragment DIGIT:  '0' .. '9';
fragment LETTER: ('a' .. 'z'|'A' .. 'Z');
```

Listing 5.11: ANTLR3-Grammatik für Wertfilter (vereinfacht)

Im Gegensatz zum Syntaxdiagramm in Abbildung 5.4 fällt hierbei für Vergleichsprädikate (*comp_pred*) die Unterscheidung zwischen linksseitigen (*value*) und rechtsseitigen (*rvalue*) Werten auf. Konkret ist für rechtsseitige Vergleichsterme die Ableitung zu Attributen oder Funktionen mit Attributen als Parametern nicht zulässig, während dies für *value* möglich ist. Ursache hierfür ist die in Abschnitt 5.1.3.3 definierte Strategie, Teile der Datentransformation im DBMS durchführen zu lassen. Die damit verbundenen Konsequenzen für die Anfrageverarbeitung werden in Abschnitt 5.2.5 näher betrachtet.

Das Parsen eines Wertfilter-Ausdrucks resultiert wie beim Infix-Aspektfilter auch hier in einem AST, für dessen Verarbeitung die als UML-Diagramm skizzierten Klassen in **Abbildung 5.7** benötigt werden. Ebenso wie die eigentliche Struktur der Wertfilter-Sprache zweistufig im Sinne prädikatenlogischer Ausdrücke aufgebaut ist, lassen sich diese Klassen zwei Gruppen zuordnen.

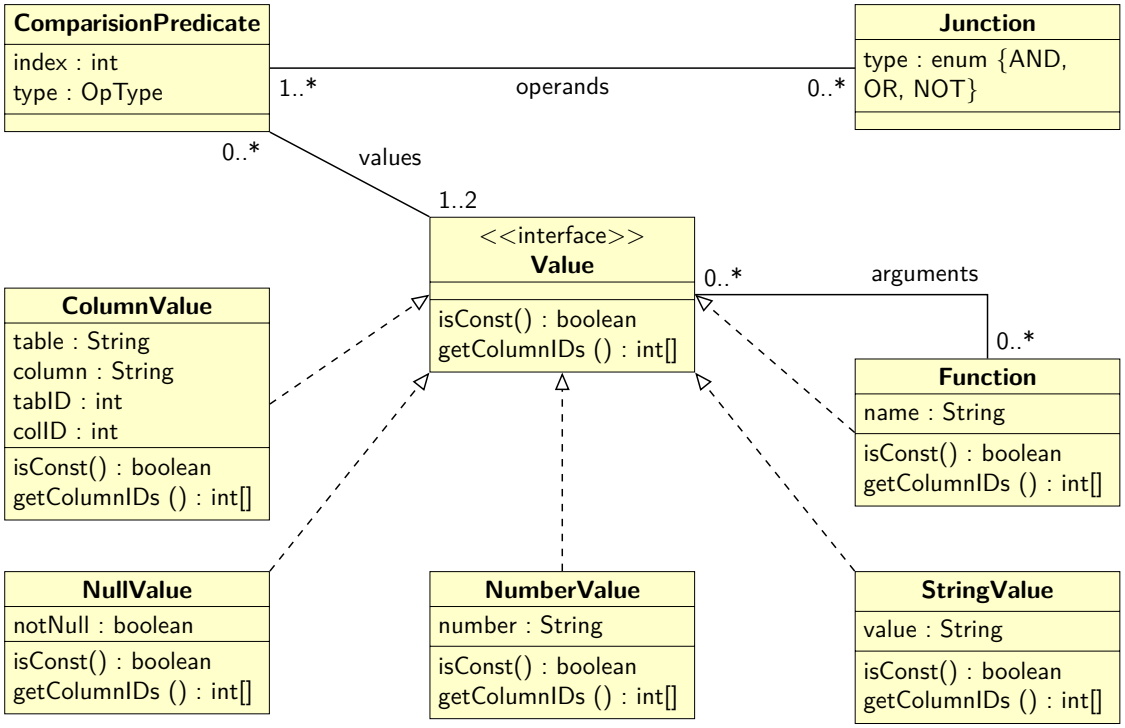


Abbildung 5.7: UML-Klassendiagramm für den Wertfilter-Syntaxbaum nach [Pie11b]

Die untere Ebene, entsprechend den Prädikatenlogik-Termen, wird durch die einzelnen das Interface **Value** implementierenden Klassen-Instanzen repräsentiert. Dabei werden Konstanten für Zahlen (**NumberValue**) und Zeichenketten (**StringValue**), Nullwerte (**NullValue**), Attribute im Sinne von Variablen (**ColumnValue**) sowie Funktionen (**Function**) mit wiederum **Value**-Instanzen als Parameter unterschieden. Zur Überprüfung der Einschränkung, dass jedes Prädikat nur von maximal einem Attribut abhängen[5] darf, dienen jeweils die Methoden *getColumnIDs* und *isConst*. Beispielsweise liefert *isConst* in **StringValue** uneingeschränkt eine wahre Aussage, während das Ergebnis jener Methode in **Function** nur wahr ist, wenn die Anwendung von *isConst* für jeden Parameter der Funktion selbst wieder eine wahre Aussage erzeugt.

Prädikate in zwei verschiedenen Formen bilden schließlich die obere Ebene. Einerseits können Vergleichsprädikate (**ComparisionPredicate**) zur Realisierung von *comp_pred* in Listing 5.11 gebildet werden, die eine Verknüpfung von **Value**-Instanzen mit ein- oder zweistelligen Operatoren auf einen Wahrheitswert abbilden. Andererseits lassen sich diese Prädikate über Junktoren der Klasse **Junction** zu komplexeren Prädikaten zusammensetzen. Verwendung finden jene Wertfilter bei den Klassen für Aspektdaten-Anfragen, auf deren Details im anschließenden Abschnitt 5.2.4 näher eingegangen wird.

[5] Hintergründe hierzu finden sich in Abschnitt 5.1.3.3 im letzten Absatz.

5.2.4 Aspektdaten-Anfragen

Nachdem in den beiden vorangegangenen Abschnitten die Metadaten-Strukturen und die Formulierung von Filterbedingungen vorgestellt wurden, sind damit die Voraussetzungen für den Zugriff auf die eigentlichen aspektspezifischen Daten gegeben. Hierfür beinhaltet die API eine interne Anfrageschnittstelle **Statement**, die sich am Interface **java.sql.PreparedStatement** in der Java Database Connectivity (JDBC) Programmierschnittstelle orientiert. Darüber hinaus findet eine auf SQL basierende Unterscheidung der Anfragetypen und der damit einhergehenden Funktionalität statt. Dementsprechend gibt es für lesende Anfragen (SELECT) das Interface **QueryStatement**, wogegen das Interface **ModifyStatement** zur Verarbeitung von modifizierenden Anweisungen (INSERT, UPDATE, DELETE) zuständig ist.

Die Zusammenhänge zwischen den genannten Schnittstellen und weiteren in diesem Kontext relevanten Interfaces sind in **Abbildung 5.8** als UML-Klassendiagramm anschaulich dargestellt. In den folgenden Abschnitten werden anfangs die Datenstrukturen **ColumnSet** und **ResultRows** genauer betrachtet. Darauf aufbauend findet eine Beschreibung der zentralen Anfrageschnittstelle sowie ihrer beiden abgeleiteten Varianten statt.

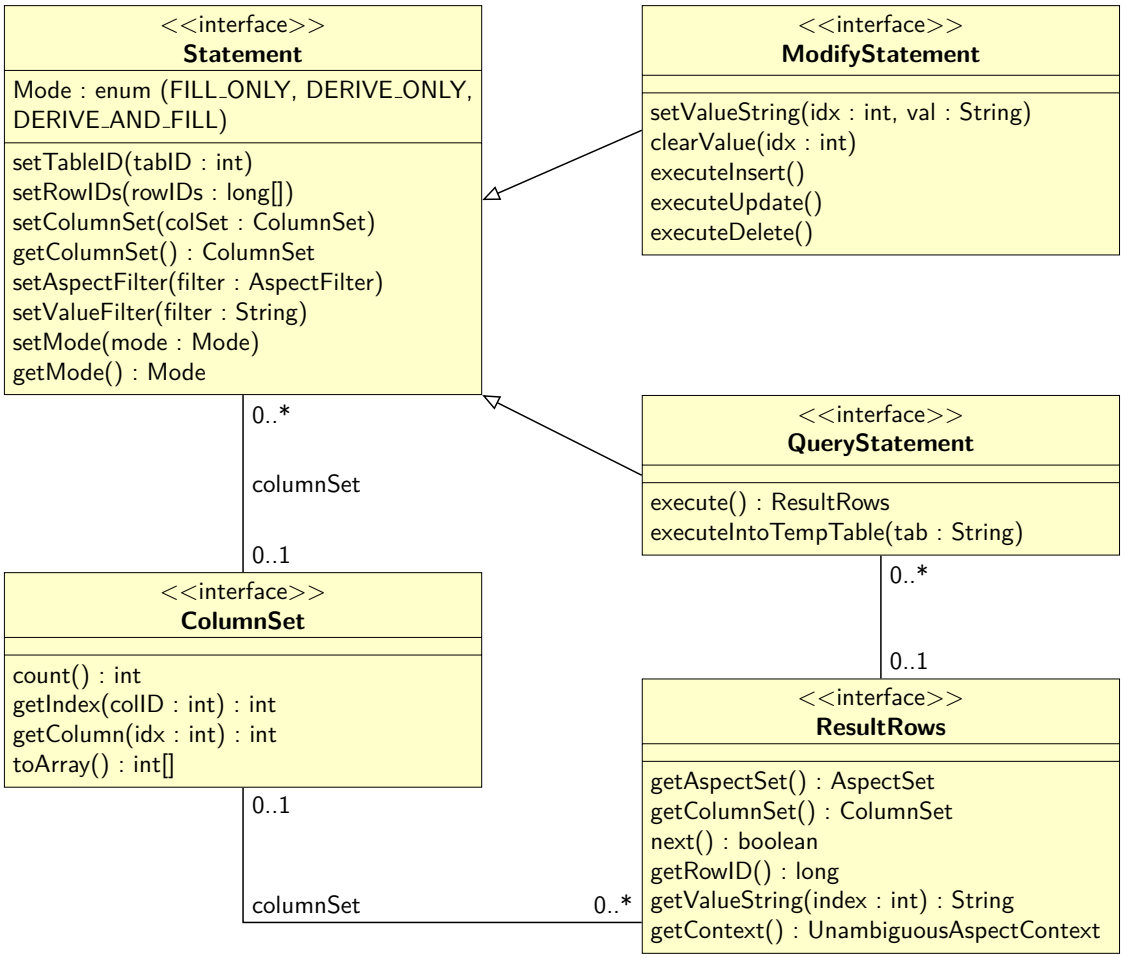


Abbildung 5.8: UML-Klassendiagramm für die Aspektdaten-Anfragen

5.2.4.1 Datenstrukturen

Zur Spezifikation und Auswertung von Anfragen spielt die Menge und Reihenfolge der relevanten Attribute eine zentrale Rolle. Analog zu den Aspektmengen in Abschnitt 5.2.3.1 wird daher die Schnittstelle **ColumnSet** entsprechend **Listing 5.12** bereitgestellt, welche effektiv die Durchnummerierung einer gewissen Attributmenge bietet. Die Instanziierung mit Hilfe der Methode *createColumnSet* in **AspectManager** erfolgt durch Übergabe einer Liste von Attributen anhand ihrer Column-IDs aus den Metadaten. Daraufhin liefert die Methode *count* die Anzahl der Attribute zurück, während *getIndex* und *getColumn* letztlich die Abfrage der Zuordnungen zwischen dem Index und der ID eines Attributs gewährleisten. Schließlich kann durch Aufruf von *toArray* wieder die ursprüngliche Liste der Column-IDs ermittelt werden.

```
interface ColumnSet {
    public int count ();
    public int getIndex (int colID);
    public int getColumn (int index);
    public int[] toArray ();
}
```

Listing 5.12: Interface ColumnSet

Im Fall einer lesenden Anfrage muss deren Ergebnis der Anwendung in geeigneter Form bereitgestellt werden. Hierfür existiert in der API das Interface **ResultRows**, welches in **Listing 5.13** wiedergegeben ist. Trotz gewisser Ähnlichkeiten zu **java.sql.ResultSet** innerhalb der JDBC-Schnittstelle gibt es im Rahmen der prototypischen Realisierung hinsichtlich des Funktionsumfangs und der Beschränkung auf eine nur lesende Cursor-Semantik deutliche Unterschiede.

```
interface ResultRows {
    public ColumnSet getColumnSet ();
    public AspectSet getAspectSet ();
    public boolean next ();
    public long getRowID ();
    public String getValueString (int columnIndex);
    public UnambiguousAspectContext getContext ();
}
```

Listing 5.13: Interface ResultRows

Jeder Datensatz ist einerseits strukturell durch die Menge seiner Attribute bestimmt, welche im zuvor erläuterten Interface **ColumnSet** verwaltet wird und über die Methode *getColumnSet* erreichbar ist. Andererseits unterliegt der aspektspezifische Inhalt den zugeordneten Aspekten, die als **AspectSet** von der Methode *getAspectSet* zurückgegeben werden. Zur Iteration über alle von der Anfrage ermittelten Datensätze im Sinne eines Cursor-Konzepts ist wiederholt die Methode *next* aufzurufen, wobei deren Rückgabewert die Verfügbarkeit weiterer Ergebniszeilen repräsentiert.

Für den aktuell unter dem Cursor stehenden Datensatz können verschiedene Informationen ausgelesen werden. Neben dem von *getRowID* gelieferten Ident bezüglich des einelementigen Schlüsselattributs in der fachlichen Basistabelle lässt sich zu jedem Attribut

mittels *getValueString* der jeweils aspektspezifische Attributwert abfragen. Hierbei ist der Index eines Attributs gemäß der zugehörigen **ColumnSet**-Ausprägung anzugeben. Schließlich liefert die Methode *getContext* den zugrunde liegenden eindeutigen Aspektkontext des gesamten Tupels.

5.2.4.2 Statement

Das in **Listing 5.14** gezeigte Interface **Statement** stellt typische Methoden zur Verfügung, die sowohl von lesenden als auch ändernden Anfragen benötigt werden.

```
interface Statement {
    public enum Mode { FILL_ONLY, DERIVE_ONLY, DERIVE_AND_FILL };
    public void setTableID (int tabID);
    public void setRowIDs (long[] rowIDs);
    public void setColumnSet (ColumnSet colSet);
    public ColumnSet getColumnSet ();
    public void setAspectFilter (AspectFilter filter);
    public void setValueFilter (String filter);
    public void setMode (Mode mode);
    public Mode getMode ();
}
```

Listing 5.14: Interface Statement

Nach der Instanziierung findet mit Hilfe verschiedener Setter-Methoden die inhaltliche Initialisierung statt. Zunächst wird über *setTableID* die fachliche Basistabelle festgelegt, auf die sich eine Anfrage beziehen soll. Weiterhin lässt sich der Wirkungsbereich bzw. Suchraum durch Übergabe der Schlüsselattributwerte an die Methode *setRowIDs* auf eine Menge von Datensätzen eingrenzen. In orthogonaler Art und Weise können auch die zu betrachtenden Attribute mittels *setColumnSet* definiert bzw. mit *getColumnSet* wieder abgefragt werden, andernfalls sind immer alle aspektabhängigen Attribute für eine Anfrage relevant. Zusätzlich sind Beschränkungen durch Aspektfilter und Wertfilter über die geeignet zu parametrisierenden Methoden *setAspectFilter* und *setValueFilter* spezifizierbar.

Da nicht zwangsläufig alle bei einer Anfrage betrachteten Attribute von der gleichen Menge funktionaler Aspekte abhängig sind, gleichzeitig aber nur ein **ColumnSet** verknüpft ist, muss die Schnittstelle ein virtuelles einheitliches Schema insbesondere hinsichtlich der Aspektsignaturen zu Grunde legen. Dadurch muss jedoch jedes einzelne zur Anfrage gehörende Tupel auf das gemeinsame Schema erweitert werden. Dies ist realisierbar über die in Abschnitt 4.2.5 beschriebenen Techniken der Auffüllung und Ableitung. Der Anwendungs-Entwickler kann die jeweilige Variante durch die Methode *setMode* festlegen, wobei diese im Fall von INSERT-Operationen ignoriert wird.

5.2.4.3 QueryStatement

Das Interface **QueryStatement** in **Listing 5.15** erweitert **Statement** für den lesenden Zugriff und wird mit Hilfe von *createQueryStatement* in **AspectManager** instanziiert. Durch die Methode *execute* wird die Anfrage ausgeführt und das Ergebnis auf einem **ResultRows**-Objekt zurückgegeben. Alternativ kann das Ergebnis auch mittels *executeIntoTempTable* in

eine temporäre Datenbanktabelle geschrieben und anschließend über reguläre SQL-Befehle prozessiert sowie mit anderen Datenbankobjekten verknüpft werden.

```
interface QueryStatement extends Statement {
    public ResultRows execute () throws AspectLookupException;
    public void executeIntoTempTable (String tempTableName) throws AspectLookupException;
}
```

Listing 5.15: Interface QueryStatement

5.2.4.4 ModifyStatement

Das Interface **ModifyStatement** in **Listing 5.16** erweitert **Statement** für den modifizierenden Zugriff und wird durch *createModifyStatement* in **AspectManager** instanziiert. Mit der Methode *setValueString* kann einem per **ColumnSet** indizierten Attribut ein (geänderter) Wert zugewiesen werden, was natürlich nur für die INSERT- und UPDATE-Operation relevant ist. Eine solche Wertzuweisung lässt sich anschließend über *clearValue* wieder zurücksetzen. Dadurch bliebe die betreffende Tabellenspalte von einer Änderungsanfrage unberührt, die dort wirkenden Aspekte würden aber weiterhin berücksichtigt.

```
interface ModifyStatement extends Statement {
    public void setValueString (int colIndex, String value);
    public void clearValue (int colIndex);
    public void executeInsert ();
    public void executeUpdate ();
    public void executeDelete ();
}
```

Listing 5.16: Interface ModifyStatement

Um einen so festgelegten Datensatz einzufügen, muss *executeInsert* gerufen werden. Hierbei kommt der gemäß Definition 4.8 spezifizierte induzierte Kontext zum Einsatz, welcher sich aus dem zuvor per *setAspectFilter* in **Statement** gesetzten Aspektfilter ergibt. Damit die in Abschnitt 4.3.2.9 formulierte Bedingung C2 gewährleistet ist, erfolgt für neue Daten implizit nur die Verknüpfung mit jenem Kontextelement, welches durch den Aufruf von *getElement(0)* an **AspectContext** resultiert. Effektiv findet dazu eine Zerlegung des Datensatzes in ein-elementige signierte Tupel statt. Dadurch lässt sich jeder dieser aspektabhängigen Attributwerte durch einen Eintrag in der Tabelle AspectValue mit entsprechenden Zuordnungen in AspectAssign darstellen. Anhand von Beispieldaten wurde dieses Prinzip in Abschnitt 4.3.3 skizziert, wobei diesbezüglich insbesondere die Abbildung 4.10 sowie Abbildung 4.11 von Interesse sind.

Schließlich kann mit *executeUpdate* eine Aktualisierung und mit *executeDelete* die Löschung von aspektspezifischen Daten durchgeführt werden. Basierend auf den einschränkenden Kriterien der zuvor festgelegten RowIDs beziehungsweise Aspekt-/Wertfilter wird hierbei im ersten Schritt die Menge der relevanten Tupel ermittelt. Dieser Vorgang entspricht weitestgehend der Verarbeitung einer lesenden Anfrage über das bereits erläuterte Interface **QueryStatement**. Im zweiten Schritt werden dann die eigentlichen Modifikationen an diesen Datensätzen vorgenommen.

5.2.5 Anfrageverarbeitung

Nach Beschreibung der zentralen Datenstrukturen und Schnittstellen innerhalb der API soll dieser Abschnitt einige Implementierungsdetails bezüglich der eigentlichen algorithmischen Prozessierung in der Zugriffsschicht näher erläutern. Grundlage hierfür ist das UML-Aktivitätsdiagramm gemäß Abbildung 5.5. Entsprechend des darin skizzierten Kontrollflusses werden die Vorverarbeitung sowohl der Wertfilter als auch der Aspektfilter im Rahmen der SQL-Generierung diskutiert. Abschließend steht noch der Aufbau der Wertetabelle beim Abrufen des Anfrageergebnisses im Fokus der Betrachtungen.

5.2.5.1 Vorverarbeitung der Wertfilter

Wie in Abschnitt 5.1.3.3 motiviert, sind die einzelnen Prädikate von Wertfilter-Ausdrücken direkt vom DBMS auszuwerten. Zu diesem Zweck wird für jedes Prädikat die SELECT-Liste der SQL-Anweisung um einen entsprechenden Eintrag ergänzt. Am Beispiel des Prädikats "*Price* $\geq$ *70*" zum Attribut *Price* ist dies in **Listing 5.17** dargestellt. Dabei resultiert dessen Spalten-ID 504 aus dem Schlüsselwert in AspectColumn nach deren exemplarischer Befüllung (Abbildung 4.8 in Abschnitt 4.3.3).

```
(CASE
    WHEN AspectValue.Column=504 THEN
        (CASE
            WHEN CAST (AspectValue.Value AS NUMERIC) >= 70.0 THEN 1
            ELSE -1
         END)
    ELSE 0
END) AS pred_i
```

Listing 5.17: SQL-Anweisung zur Auflösung von Wertfilter-Prädikaten

Generell kann ein Prädikat erfüllt oder nicht erfüllt sein, wobei diese booleschen Werte als 1 und -1 codiert werden. Darüber hinaus wird der Wert 0 zurückgegeben, wenn das aktuell verarbeitete Attribut nicht mit dem Attribut im Prädikat übereinstimmt. Dieser Fall ist zu berücksichtigen, da alle aspektspezifischen Attribute gleichzeitig in der Spalte Column von Tabelle AspectValue gespeichert sind.

Für die weitere Verarbeitung dieser Prädikat-Auswertungen würde theoretisch die Speicherung aller Ergebnisse (-1, 0, 1) in einem Integer-Array ausreichen, wobei der Wert des Prädikats pred_i an i-ter Stelle zu finden wäre. Zur Optimierung bezüglich Speicherplatz und Performance insbesondere bei umfangreicheren Wertfilter-Ausdrücken erfolgt jedoch die Ablage mittels zweier Bitlisten, wobei eine Liste für den booleschen Prädikatwert selbst zuständig ist und und die zweite Liste über dessen Gültigkeit entscheidet.

5.2.5.2 Vorverarbeitung der Aspektfilter

Zwar ist wie in Abschnitt 5.1.3.2 beschrieben die Auswertung eines Aspektfilters im Allgemeinen erst auf Basis des Anfrageergebnisses vollständig durchführbar. Allerdings kann versucht werden, diese Datenmenge soweit wie möglich einzuschränken. Dazu identifiziert

die Schnittstelle solche Ausdrücke im Aspektfilter, die bei der späteren Auswertung ohnehin den Ausschluss gewisser Daten erzwingen. Jene Bedingungen werden bei der Generierung der SQL-Anfrage zusätzlich berücksichtigt.

Die hier vorgestellte Referenz-Implementierung realisiert eine derartige Analyse des Aspektfilter-Ausdrucks mit Hilfe der *Bottom-Up-Technik*. Dazu wird zu jedem funktionalen Aspekt die Menge der zulässigen oder unzulässigen Aspektausprägungen bestimmt. Dabei muss diese Entscheidung unabhängig von jeglichen anderen Aspekten sein, da der vom DBMS durchzuführende Verbund zwischen AspectValue und AspectAssign nur Tupel erzeugt, die genau eine Aspektschlüsselwert-Zuordnung beinhalten (siehe Festlegung zur Aufgabenteilung in Abschnitt 5.1.3.1). Komplexe logische Verknüpfungen lassen sich zu diesem Zeitpunkt also noch nicht entscheiden.

Anhand eines Beispiels soll das Prinzip dieser Vorverarbeitung verdeutlicht werden. Dazu sei folgender Aspektfilter-Ausdruck gegeben:

(*'Version'* = *'15'* `OR` *'Version'* `IS NULL`) `AND NOT` (*'Region'* `IN` (*'AUT','GER','SUI'*))

Für die beiden Aspekte *Version* und *Region* wurden jeweils eigenständige Bedingungen formuliert und mit `AND` verknüpft. Daraus ergeben sich durch die oben beschriebene Analyse folgende Mengen zulässiger bzw. unzulässiger Aspektausprägungen:

$$\begin{aligned} \textit{'Version'} &\mapsto \{\textit{'15'}, \texttt{NULL}\} \\ \textit{'Region'} &\not\mapsto \{\textit{'GER'},\ \textit{'AUT'},\ \textit{'SUI'}\} \end{aligned}$$

Die Anfrage-Generierung kann also bereits um eine WHERE-Klausel erweitert werden, sodass aspektspezifische Attributwerte einerseits der Version *'15'* oder keiner Version zugeordnet sein müssen und andererseits nicht für die Regionen *'GER'*, *'AUT'* und *'SUI'* gelten dürfen.

5.2.5.3 Aufbau der Wertetabelle

Nach Ausführung der Datenbankanfrage liefert das DBMS alle Tupel aus dem Verbund von AspectValue und AspectAssign, sofern sie nicht durch die zuvor diskutierten Vorverarbeitungen bereits gefiltert wurden. Wie in Abschnitt 5.1.3.1 aufgezeigt, ist dieses Ergebnis noch unsortiert, weswegen die zur Pivotisierung notwendige Gruppierung nach RowID und Aspektausprägung durch die Zugriffsschicht erfolgen muss. Dazu wird zunächst für jede RowID eine separate Wertetabelle mit den Zuordnungen der Aspektausprägungen zu jedem aspektspezifischen Attributwert erzeugt.

Nach dem Einlesen aller Ergebnistupel enthalten die aufgebauten Wertetabellen jeweils inhaltlich alle Aspektzuordnungen zu einem Attributwert. Dadurch ist deren Pivotisierung gemäß Abbildung 4.12 (Schritt i) abgeschlossen. Auf Grundlage der nun vorliegenden Aspektsignaturen kann eine Hashtabelle aufgebaut werden, welche für den anschließenden zweiten Pivotisierungsschritt zum Einsatz kommt. Anhand der Aspektsignaturen lassen sich nun auch gegebenenfalls definierte Aspektfilter auswerten und somit nicht vereinbare Signaturen verwerfen. Alle danach noch gültigen Attributwerte mit der identischen RowID und Signatur werden schließlich zu uniformen Tupeln gruppiert.

5.3 Verwendung der API

Im letzten Teil dieses Kapitels sollen die Funktionsweise und sowie die Anwendbarkeit der in Abschnitt 5.2 eingeführten Schnittstellen-Strukturen anhand exemplarischer Code-Fragmente demonstriert werden. Diese beziehen sich sowohl auf das Anwendungsbeispiel aus Abschnitt 3.2 als auch auf die dazu in Abschnitt 4.3.3 beispielhaft erzeugten Daten.

5.3.1 Zugriff auf den Aspektkatalog

Der Einstieg in die API der Zugriffsschicht erfolgt auf Grundlage des in Abschnitt 5.2.1 beschriebenen **AspectManager**. Zu dieser Schnittstelle existiert eine implementierungsabhängige Instanz, an der sich wiederum ein **AspectCatalogManager** für den Zugriff auf die Aspektkatalogdaten instanziieren lässt. Konkret werden die Idents der Aspekte Sprache und Markt inklusive der zugehörigen Schlüsselwerte in **Listing 5.18** abgefragt (Zeile 5 bis 11). Zusätzlich sind die IDs der Tabelle TEST.MODULE sowie für die Attribute NAME und PRICE zu ermitteln, da sich alle weiteren Anfragen in den nächsten Abschnitten auf jene Tabelle beziehen.

```
AspectManager am = /* Verweis auf AspectManager-Instanz holen ... */
AspectCatalogManager acm = am.getAspectCatalogManager ();
int aspLang = acm.lookupAspectID ("Sprache");
int aspLangEn = acm.lookupAspectKeyValueID (aspLang, "en");
int aspMarket = acm.lookupAspectID ("Markt");
int aspMarketUS = acm.lookupAspectKeyValueID (aspMarket, "USA");
int aspMarketUK = acm.lookupAspectKeyValueID (aspMarket, "UK");
int aspMarketEU = acm.lookupAspectKeyValueID (aspMarket, "EUR");
int tabID = acm.lookupTableID ("Test", "Module");
int colNameID = acm.lookupColumnID (tabID, "Name");
int colPriceID = acm.lookupColumnID (tabID, "Price");
```

Listing 5.18: Code-Beispiel für den Aspektkatalog-Zugriff

5.3.2 Lesen von Aspektdaten

Der lesende Zugriff auf aspektspezifische Daten in der fachlichen Tabelle MODULE ist durch **Listing 5.19** beispielhaft skizziert und wird nachfolgend erläutert. An dem bereits in Listing 5.18 instanziierten **AspectManager** lässt sich nun auch eine Anfrageinstanz erzeugen. Aus der über tabID referenzierten Tabelle MODULE sollen dabei für zwei konkrete Datensätze (18744, 21073) die Attribute NAME und PRICE abgefragt werden. Weiterhin wird über einen Aspektfilter auf diejenigen Attributwerte eingeschränkt, die der englischen Sprache und wahlweise dem Markt USA oder UK zugeordnet sind (Zeile 4 bis 9). Die zusätzliche Abhängigkeit beider Attribute vom Aspekt Version nach der Festlegung in Abbildung 3.5 sei hier nicht weiter betrachtet.

Durch Aufruf der *execute*-Methode wird die zuvor spezifizierte Anfrage auf der Datenbank ausgeführt (Zeile 11). Anschließend lässt sich mit einer Schleife schrittweise über alle Ergebniszeilen des **ResultRows**-Objekts iterieren (Zeile 18). Um dabei die aspektspezifischen Attributwerte sowie die in diesem Kontext gültigen Aspektausprägungen extrahieren zu können (Zeile 20 bis 22), werden die Indizes der Attribute und Aspekte benötigt. Hierfür

```
1  QueryStatement st = am.createQueryStatement (tabID);
   st.setRowIDs (new long[] {18744, 21073});
3  st.setColumnSet (am.createColumnSet (new int[] {colNameID, colPriceID}));
   st.setAspectFilter (am.createAspectFilter (
5    AspectFilter.and (
       AspectFilter.keyValue (aspLang, aspLangEn),
7      AspectFilter.or (
         AspectFilter.keyValue (aspMarket, aspMarketUS),
9        AspectFilter.keyValue (aspMarket, aspMarketUK)))));

11 ResultRows result = st.execute ();
   AspectSet aSet = st.getAspectSet ();
13 int aspMarketIdx = aSet.getIndex (aspMarket);
   ColumnSet cSet = st.getColumnSet ();
15 int colNameIdx = cSet.getIndex (colNameID);
   int colPriceIdx = cSet.getIndex (colPriceID);
17
   while (result.next ()) {
19   long rowID = result.getRowID ();
     String name = result.getValueString (colNameIdx);
21   String price = result.getValueString (colPriceIdx);
     int marketID = e.getIndexKeyValue (aspMarketIdx);
23   AspectContextElement e = result.getContext ().getElement ();
   } /* verknüpfe Daten anhand von RowID mit den Daten des Kernaspektes
```

Listing 5.19: Code-Beispiel für das Lesen von Aspektdaten

sind die zum aktuellen **QueryStatement** zugehörigen Ausprägungen von **AspectSet** und **ColumnSet** mitels der *getIndex*-Methoden abzufragen (Zeile 12 bis 16). Der generell eindeutige Aspektkontext zu einer Ergebniszeile spielt für die Anwendung immer dann eine Rolle, wenn je RowID nicht nur ein Ergebnistupel im Anfrageergebnis vorliegt.

5.3.3 Ändern von Aspektdaten

Das Verändern aspektspezifischer Daten in der Tabelle MODULE ist in **Listing 5.20** beispielhaft skizziert und wird nachfolgend erläutert.

```
   ModifyStatement st = am.createModifyStatement (tabID);
2  ColumnSet colSet = am.createColumnSet (new int[] {colName, colPrice});
   int colNameIdx = colSet.getIndex (colName);
4  int colPriceIdx = colSet.getIndex (colPrice);
   st.setColumnSet (colSet);
6  st.setRowIDs (new long[] {18744});
   st.setAspectFilter (am.createAspectFilter (
8    AspectFilter.or (
       AspectFilter.keyValue (aspMarket, aspMarketEUR),
10     AspectFilter.keyValue (aspLang, aspLangDE))));
   st.setValueString (colNameIdx, "Flachkopf-Schraube M5");
12 st.setValueString (colPriceIdx, "0.42");
   st.executeUpdate ();
```

Listing 5.20: Code-Beispiel für das Ändern von Aspektdaten

Das Vorgehen, um bereits vorhandene Datensätze zu manipulieren, hat große Ähnlichkeit zum Einfügen neuer Daten und erfolgt über das am **AspectManager** zu instanziierende **ModifyStatement**. Mit dem Ziel, die Attributwerte von NAME und PRICE in der Tabelle MODULE anzupassen, wird nun das entsprechende **ColumnSet** erzeugt und am Statement-Objekt gesetzt (Zeile 2 und 6). Durch Festlegung der RowID auf 18744 ist eindeutig das betreffende Tupel identifiziert. Der implizit durch einen Aspektfilter definierte Aspektkontext schränkt die nachfolgenden Änderungen auf den deutschsprachigen Teil des EU-Marktes ein (Zeile 8 bis 11) und der Versions-Aspekt soll wiederum keine Rolle spielen. Nachdem mittels *setValueString* die relevanten Attribute mit den entsprechenden Werten belegt wurden, kann jene Änderung ausgeführt werden. Dabei werden jedoch in den Tabellen ASPECTASSIGN und ASPECTVALUE keine neuen Zeilen hinzugefügt. Stattdessen findet lediglich eine Änderung existierender Werte in ASPECTVALUE.VALUE statt, diese allerdings abhängig von den gesetzten Einschränkungen bezüglich Aspektkontext und RowIDs mehr oder weniger umfangreich.

5.3.4 Einfügen von Aspektdaten

Das Hinzufügen aspektspezifischer Daten in die Tabelle MODULE ist in **Listing 5.21** beispielhaft skizziert und wird nachfolgend erläutert.

```
1  ModifyStatement st = am.createModifyStatement (tabID);
   UnambiguousAspectContext ctx = am.createUnambiguousAspectContext (
3    am.createAspectSet (acm.getColumnDependencies (colPriceID)));
   int aspMarketIdx = ctx.getAspectSet ().getIndex (aspMarket);
5  ColumnSet colSet = am.createColumnSet (new int[] {colPriceID});
   int colPriceIdx = colSet.getIndex (colPriceID);
7
   st.setColumnSet (colSet);
9  st.setRowIDs (new long[] {21073});

11 ctx.getElement ().setIndexKeyValue (aspMarketIdx, aspMarketUS);
   st.setAspectContext (ctx);
13 st.setValueString (colPriceIdx, "1.29");
   st.executeInsert ();
15 ctx.getElement ().setIndexKeyValue (aspMarketIdx, aspMarketUK);
   st.setAspectContext (ctx);
17 st.setValueString (colPriceIdx, "0.79");
   st.executeInsert ();
19 ctx.getElement ().setIndexKeyValue (aspMarketIdx, aspMarketEU);
   st.setAspectContext (ctx);
21 st.setValueString (colPriceIdx, "0.95");
   st.executeInsert ();
```

Listing 5.21: Code-Beispiel für das Einfügen von Aspektdaten

Neue Datensätze lassen sich über die API nur als einzelne Tupel je Anweisung einfügen. Dazu wird eine Instanz von **ModifyStatement** benötigt, die unter Angabe der Tabellen-ID für MODULE am **AspectManager** erzeugt werden kann. Im vorliegenden Beispiel sollen nur zum Attribut PRICE neue aspektspezifische Daten eingefügt werden. Dementsprechend bilden die darauf wirkenden Aspekte (*getColumnDependencies* (colPriceID)) die Grundlage

zur Generierung eines eindeutigen Aspektkontexts (Zeile 2), welcher nach Abschnitt 5.2.4.4 eine notwendige Voraussetzung für neu einzufügende Datensätze darstellt.

Nach Festlegung des fachlichen Datensatzes mit der RowID 21073 werden jeweils für die Ausprägungen USA, UK und EUR des Aspekts Markt drei individuelle Preisangaben dem Attribut PRICE zugewiesen (Zeile 13, 17 und 21). Dabei sollen diese vom ebenfalls wirkenden Aspekt Version nicht abhängig sein, weswegen der erzeugte eindeutige Aspektkontext neben dem Aspektschlüsselwert für den Markt eine NULL-Belegung ($\perp$) für die Version enthält. Anschließend lassen sich die Informationen mit der Methode *executeInsert* in die Datenbank schreiben (Zeile 14, 18 und 22). Die Methode muss jedoch mit einer Ausnahme abbrechen, falls zum angegebenen Tupel und Kontext bereits ein Attributwert existiert.

5.3.5 Löschen von Aspektdaten

Das Löschen aspektspezifischer Daten aus der Tabelle MODULE ist in **Listing 5.22** beispielhaft skizziert und wird nachfolgend schrittweise erläutert.

```
ModifyStatement st = am.createModifyStatement (tabID);

/* lösche alle US-amerikanischen Preise */
st.setColumnSet (am.createColumnSet (new int[] {acm.lookupColumnID (tabID, "Price")}));
st.setAspectFilter (am.createAspectFilter (AspectFilter.keyValue (aspMarket, aspMarketUS)));
st.executeDelete ();
```

Listing 5.22: Code-Beispiel für das Löschen von Aspektdaten

Analog zur Änderungs-Operation können auch durch einen Aufruf der Methode *exeuteDelete* am **ModifyStatement** mehrere Tupel betroffen sein. Die Reichweite einer solchen Anweisung ist dabei direkt abhängig von den zuvor gesetzten Einschränkungen. Im konkreten Beispiel sollen die US-amerikanischen Preise aller Einträge der Tabelle MODULE entfernt werden. Einerseits wird dazu das **ColumnSet** nur aus dem Attribut PRICE aufgebaut (Zeile 4). Andererseits definiert sich der Aspektfilter über die Bedingung, dass am Aspekt Markt der Wert USA gesetzt sein muss (Zeile 5). Generell hat eine solche Lösch-Operation jedoch weder Einfluss auf die aspektunabhängigen Daten der Fachtabelle noch auf die Aspektabhängigkeiten, d.h. die Aspekte Markt und Version beeinflussen weiterhin das Attribut PRICE.

5.3.6 Fazit

Anhand einer prototypischen Implementierung wurde im Verlauf des Kapitels die prinzipielle Funktionsfähigkeit des API-Ansatzes als Zugriffstechnik auf das Referenzmodell gezeigt und mit Hilfe von beispielhaften Anwendungsfällen in den letzten Abschnitten demonstriert. Im folgenden Kapitel soll abschließend der Nachweis für die effiziente Verwendbarkeit der API im Rahmen eines Performancetests erfolgen.

Kapitel 6

Test und Bewertung

Nachdem mit Kapitel 5 die prinzipielle Funktionsfähigkeit der Zugriffsschicht im Rahmen einer prototypischen Implementierung gezeigt wurde, soll nun deren Praxistauglichkeit unter realistischen Bedingungen validiert werden. Basierend auf [Pie11b] erfolgt dies mit Hilfe adäquater Performancetests. Das dabei zugrunde gelegte Szenario wird einleitend in **Abschnitt 6.1** beschrieben. Anschließend findet in **Abschnitt 6.2** die Festlegung der repräsentativen Menge von Beispielanfragen statt. Mit der so definierten Workload lässt sich neben der Zugriffsschicht auch die unmittelbare SQL-Schnittstelle des DBMS als weiterer Testkandidat konfrontieren, worauf **Abschnitt 6.3** detailliert eingeht. Die Durchführung der einzelnen Testreihen sowie deren Ergebnisse werden schließlich in **Abschnitt 6.4** dargestellt, bevor eine Auswertung jener Daten in **Abschnitt 6.5** das Kapitel abrundet.

6.1 Szenario

Zur Durchführung des Performancetests ist eine entsprechende Testumgebung notwendig, welche primär durch die in Abschnitt 6.1.1 definierte Hardware und Architektur gekennzeichnet ist. Darüber hinaus hat die eigentliche Anwendung mit dem in Abschnitt 6.1.2 beschriebenen Datenmodell ebenfalls direkten Einfluss auf die inhaltliche Ausprägung der Test-Parameter. Hierbei spielt auch die Verfügbarkeit geeigneter Testdaten eine wichtige Rolle, auf deren künstliche Erzeugung in Abschnitt 6.1.3 eingegangen wird.

6.1.1 Testumgebung

Grundlage der eingesetzten Testumgebung ist ein handelsüblicher Desktop-PC mit einem Doppelkernprozessor vom Typ AMD Athlon 64 X2 3800+ [AMD07] inklusive 3GB Arbeitsspeicher. Die Verwaltung jener Hardware übernimmt das Betriebssystem 64-Bit Debian/GNU Linux 7.0 (Kernelversion 2.6.32). Zur Abbildung des relationalen Referenzmodells kommen die beiden Datenbanksysteme DB2 LUW 9.7 Express-C [CHA09] und PostgreSQL 9 [PGD13] zum Einsatz, um eine belastbare Vergleichbarkeit und Bewertung der Ergebnisse gewährleisten zu können. Schließlich ist eine Java Virtual Machine (Oracle HotSpot 64-Bit Server VM) installiert, welche für die Ausführung der zu testenden Zugriffsschicht-Implementierung benötigt wird.

6.1.2 Datenmodell

Als fachlicher Hintergrund für den Performancetest dient hier wieder das in Abschnitt 3.2 eingeführte Anwendungsbeispiel zur Verwaltung von Produktkomponenten mit Hilfe der Tabelle MODULE. Den grundsätzlichen Aufbau zeigt **Abbildung 6.1**, das zugehörige SQL-Skript ist in **Listing C.1** im Anhang dargestellt.

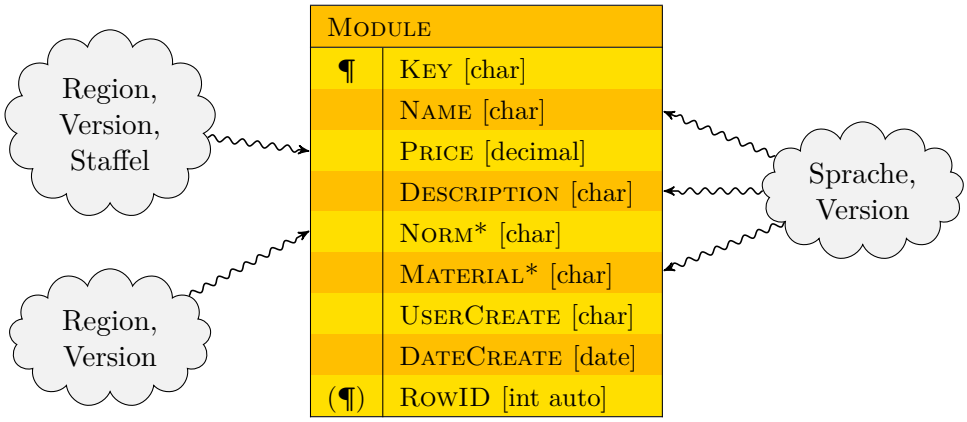


Abbildung 6.1: Tabelle MODULE und funktionale Aspekte

Weiterhin stehen einige der Attribute unter dem Einfluss von den funktionalen Aspekten Sprache, Region, Version und Staffel, wie sie bereits anschaulich in Abschnitt 3.2.2 beschrieben wurden. Die daraus resultierenden aspektspezifischen Informationen verwaltet das relationale Referenzmodell der AOD gemäß Abbildung 4.3. Mit Hilfe des SQL-Skripts in **Listing C.2** im Anhang lassen sich die zugehörigen Tabellen-Strukturen erzeugen.

6.1.3 Testdaten

Ausgehend von dem zuvor erläuterten Datenmodell definiert sich die Datenbasis für den Performancetest aus den eigentlichen Informationen in der Fachtabelle MODULE, deren aspektspezifischen Wertzuordnungen in den Tabellen ASPECTVALUE und ASPECTASSIGN sowie den Aspektkatalogdaten im Referenzmodell. Dabei sind alle Metadaten, wie beispielsweise die Menge der funktionalen Aspekte und Aspektausprägungen, vom Inhalt in MODULE sowie den aspektspezifischen Werten unabhängig und in ihrem Umfang relativ gering. Deshalb werden sie durch das SQL-Skript in **Listing C.3** im Anhang angelegt.

Demgegenüber sind für realistische Testbedingungen umfangreichere Nutzdaten in den anderen oben genannten Tabellen notwendig, wobei sich aspektspezifische Ausprägungen in ASPECTVALUE zwingend auf existierende Datensätze in MODULE beziehen müssen. Zur Vermeidung des daraus resultierenden hohen Aufwands und Fehlerpotentials bei der manuellen Testdaten-Erzeugung wurde hierfür das Java-Programm **RandomFillUniform** entwickelt. Nachfolgend steht dessen Verwendung im Vordergrund, eine detaillierte Beschreibung der Funktionsweise ist in [Pie11b] zu finden.

Die Generierung von Testdaten mit Hilfe von **RandomFillUniform** lässt sich inhaltlich über einige Parameter beeinflussen. Das wichtigste Kriterium hierbei ist die Vorgabe der Kardinalität von MODULE. Dadurch wird automatisch auch die Grundlage für die Menge der Einträge in den Tabellen ASPECTVALUE und ASPECTASSIGN definiert. Deren Umfang

ist darüber hinaus abhängig von einem zweiten Parameter für die mittlere Anzahl aspektspezifischer Daten je Attribut. Die tatsächliche Datenmenge ergibt sich dann aus einer pseudozufälligen Verteilung unter Beachtung der verfügbaren Schlüsselwerte je Aspekt. Schließlich steuert ein dritter Parameter den Anteil jener Daten in AspectValue und AspectAssign, welche in der Praxis durch weitere Fachtabellen unter Aspekteinfluss verursacht würden. Im aktuellen Testszenario erfolgt deren Simulation durch künstliche Blinddaten für die Platzhalter-Relation Aux gemäß ihrer Metadaten in Listing C.3.

Mit Hilfe dieser Parametrisierung der Testdaten-Generierung können nun sehr einfach verschiedene Datenbestände erzeugt werden, um Stärken und Schwächen der einzelnen in Abschnitt 6.3 vorgestellten Testkandidaten aus den Ergebnissen ableiten zu können. Konkret kommen für den Test die in **Tabelle 6.1** aufgeführten Datenbestände zum Einsatz. Dabei sind jeweils die absoluten Kardinalitäten der Tabellen Module und AspectValue sowie die durchschnittliche Anzahl ($\varnothing$) aspektspezifischer Werte je Attribut, die zugehörige Standardabweichung (σ) von diesem Mittelwert und der Anteil von Blinddaten in der Tabelle AspectValue angegeben. Sowohl die Ausprägung der aspektspezifischen Attributwerte als auch deren Zuordnung zu den Schlüsselwerten relevanter Aspekte unterliegt dabei jener pseudozufälligen Verteilung.

Name	|Module|	|AspectValue|	$\varnothing \frac{\text{asp. Werte}}{\text{Attribut}}$	$\sigma \frac{\text{asp. Werte}}{\text{Attribut}}$	Blinddaten %
DS01	138	3743	5.4	4.6	54
DS02	503	20979	8.3	7.5	62
DS03	1977	144245	14.6	12.1	60

Tabelle 6.1: Übersicht zu den verwendeten Testdatenbeständen

6.2 Workload

Basierend auf dem zuvor in Abschnitt 6.1.2 festgelegten Anwendungsbeispiel können nun die eigentlichen Anfragen als Workload für den Test spezifiziert werden. Um hierbei möglichst alle praxisrelevanten Anwendungsfälle abzudecken, findet zunächst deren Klassifikation bezüglich inhaltlicher Kriterien in Abschnitt 6.2.1 statt. Anschließend erfolgt die umgangssprachliche Formulierung und Klassifizierung der konkreten Anfragen im Abschnitt 6.2.1.

6.2.1 Klassifikation der Anfragen

Eine Anfrage im Kontext des relationalen Referenzmodells für die AOD lässt sich nach den im Folgenden erläuterten Dimensionen klassifizieren. Dabei sollen für den Performancetest generell nur Anfragen zur Verarbeitung aspektspezifischer Daten betrachtet werden.

Wirkungsbereich: Dieses Kriterium dient der Unterscheidung, ob für eine Anfrage die aspektspezifischen Daten nur eines fachlichen Objekts relevant (**I**ndividuell) oder mehrere fachliche Objekte involviert sind (**M**engenorientiert).

Zugriffsart: Abhängig von der Art des Datenzugriffs können analog zu SQL lesende (**S**elect), einfügende (**I**nsert), ändernde (**U**pdate) und löschende (**D**elete) Anfragen unterschieden werden.

Aspektfilter: Gemäß der in Abschnitt 5.1.2.1 spezifizierten Ausdrucksmöglichkeit kann die Festlegung der für eine Anfrage gültigen Aspektausprägungen eindeutig (**1**), mehrdeutig (+) oder vollkommen unbeschränkt (*) sein.

Wertfilter: Gemäß der in Abschnitt 5.1.2.2 spezifizierten Ausdrucksmöglichkeit kann die Festlegung der für eine Anfrage zulässigen Attributwerte eindeutig (**1**), mehrdeutig (+) oder vollkommen unbeschränkt (*) sein.

Mit Hilfe der in **Abbildung 6.2** dargestellten Ordnung aller vier Dimensionen ergibt sich für jede Anfrage aufgrund ihrer Bewertung in jener Klassifikation ein symbolischer Name bestehend aus vier Zeichen. Zusätzlich wird dieser symbolische Präfix um eine fortlaufende Nummer ergänzt, wobei damit explizit keine Ausführungsreihenfolge verbunden ist. Dadurch lässt sich bereits aus der Benennung einer Anfrage sehr einfach deren Charakteristik ableiten. Auf Grundlage dieser Systematik erfolgt nun im nächsten Abschnitt die eigentliche Spezifikation der Testanfragen.

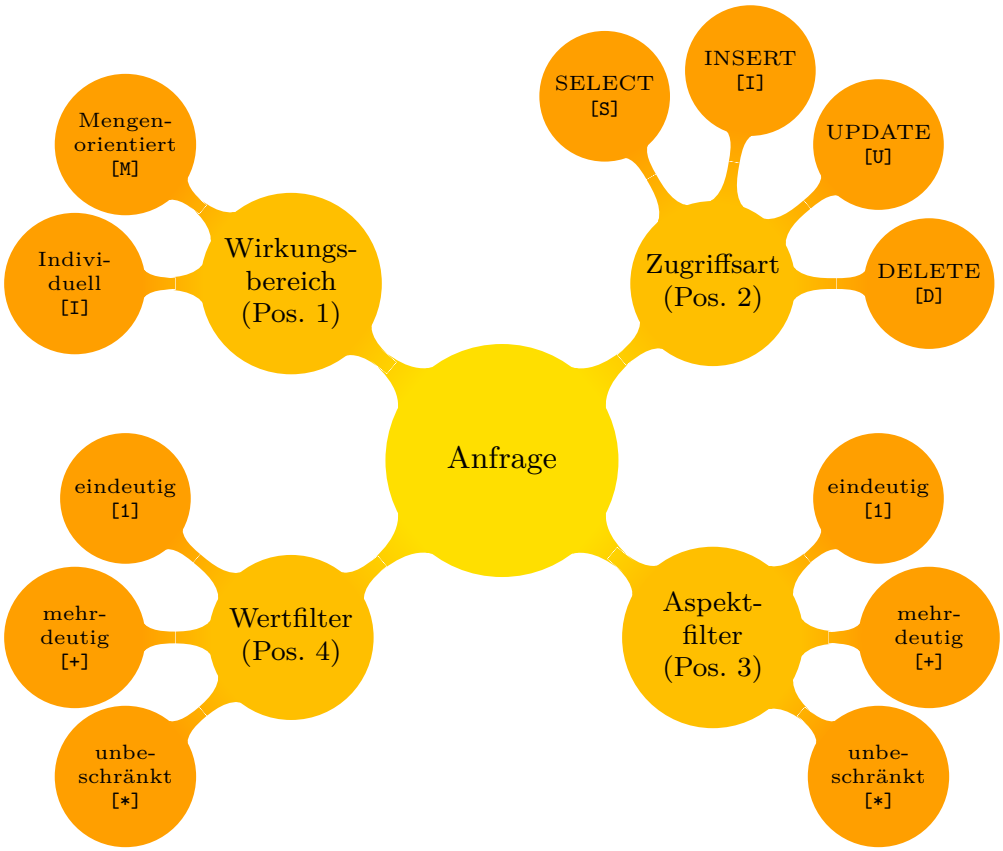


Abbildung 6.2: Klassifikation der Testanfragen

6.2.2 Spezifikation der Anfragen

Von den vielfältigen Kombinationen innerhalb der vorgestellten Klassifikation nach Abbildung 6.2 sollen für die Testanfragen nur jene Varianten betrachtet werden, die sich aus typischen Nutzungsszenarien im Anwendungsbeispiel gemäß Abbildung 6.1 ergeben.

Nachfolgend sind derartige Anfragen in natürlicher Sprache formuliert. Die tatsächliche Umsetzung erfolgt in Abschnitt 6.3 spezifisch für die einzelnen Testkandidaten.

`IS**01`: Ermittle zu einer konkreten Produktkomponente alle aspektspezifischen Daten.

`IS+*02`: Ermittle zu einer konkreten Produktkomponente alle aspektspezifischen Daten, die nur für Frankreich relevant sind.

`II1103`: Füge zu einer konkreten Produktkomponente eine Übersetzung in englischer Sprache für den Namen, die Bezeichnung und das Material sowie einen Preis in britischen Pfund ein. Hierbei ist sowohl die aktuelle Version (entspricht einem $\perp$-Wert) als auch die Preisstaffel 'standard' zu nutzen.

`ID1*04`: Lösche zu einer konkreten Produktkomponente den Preis, der im Euroraum in Version 3 für die Preisstaffel 'standard' hinterlegt ist.

`IU1105`: Setze zu einer konkreten Produktkomponente den Preis auf 0.44 für die Preisstaffel 'standard' im Euroraum und archiviere den bisherigen Preis unter einer neuen Version.

`MS**06`: Ermittle zu allen Produktkomponenten alle aspektspezifischen Daten.

`MS++07`: Ermittle Name, Beschreibung und Preis aller Produktkomponenten, deren Preis zur Preisstaffel 'standard' im Euroraum zwischen 0.5 und 1.0 liegt.

`MD**08`: Lösche alle aspektspezifischen Daten aller Produktkomponenten.

`MD+*09`: Lösche alle aspektspezifischen Daten aller Produktkomponenten, denen die Version 1 oder 2 zugewiesen sind.

6.3 Testfeld

Nachdem im vorherigen Abschnitt 6.2.2 die Anfragen für den Test aus Anwendungssicht beschrieben wurden, kann nun deren Umsetzung mit Hilfe der Referenzimplementierung in Abschnitt 6.3.1 erfolgen. Wie zu Beginn des Kapitels erwähnt, soll darüber hinaus für eine qualifizierte Bewertung der Zugriffsschicht-API jene Workload auch durch die standardisierte SQL-Schnittstelle als Vergleichskandidat beantwortet werden. Dazu kommen einerseits die in Abschnitt 6.3.2 vorgestellten generischen Sichten (gSQL) und andererseits manuell optimierte spezifische Anweisungen (sSQL) gemäß Abschnitt 6.3.3 zum Einsatz.

6.3.1 Zugriffsschicht (API)

Bezugnehmend auf die in Abschnitt 5.2 spezifizierte API illustrieren die nachfolgenden Code-Fragmente die Realisierung der Anfragen. Dabei wurden für eine bessere Lesbarkeit und kompakte Darstellung zahlreiche Details ausgeblendet, welche zum Aufruf aus einer Anwendung heraus notwendig sind. Diese Informationen sind dem vollständigen Code der Java-Klasse **TestRunnableRefImpl** in [Pie11b] zu entnehmen. Insbesondere referenziert die Variable „am“ ein **AspectManager**-Objekt, welches die zentrale Verwaltungsinstanz gemäß Abschnitt 5.2.1 darstellt.

Für eine effektive Realisierung der Anfrage `IS**01` sind die elementaren API-Aufrufe in **Listing 6.1** dargestellt. Im ersten Schritt wird basierend auf der Fachtabelle MODULE eine Instanz von **QueryStatement** für lesende Anfragen erzeugt. Die Beschränkung des Suchraums auf eine konkrete Produktkomponente erfolgt durch Festlegung auf deren zugeordnete rowid. Anschließend wird die Anfrage ausgeführt. Da jedoch auch der Transport der Ergebnistupel in das Anwendungsprogramm einen Einfluss auf das zeitliche Verhalten hat, simuliert im zweiten Schritt eine Iteration über das **ResultRows**-Objekt diesen Vorgang im Rahmen des Performancetests.

```
int tabID = am.getAspectCatalogManager ().lookupTableID (caSchema, "Module");
QueryStatement qs = am.createQueryStatement (tabID);
qs.setRowIDs (new long [] {rowid});

ResultRows rr = qs.execute ();
while (rr.next ()) {
  rr.getRowID ();
  AspectContextElement e = rr.getContext ().getElement ();
  for (int i = 0; i < as.count (); i++) {
    e.getIndexKeyValue (i);
  }
  for (int i = 0; i < cs.count (); i++) {
    rr.getValueString (i);
  }
}
```

Listing 6.1: Umsetzung der Anfrage `IS**01` mittels API

Für eine effektive Realisierung der Anfrage `IS+*02` sind die elementaren API-Aufrufe in **Listing 6.2** dargestellt. Aufgrund der inhaltlichen Nähe zur Anfrage `IS**01` lässt sich auch ein großer Teil jener Anweisungen wieder verwenden. Zusätzlich erfolgt vor Ausführung der Methode *execute* am **QueryStatement**-Objekt noch die Definition eines Aspektfilters, um auf aspektspezifische Werte mit Bezug zu Frankreich einzuschränken.

```
int tabID = am.getAspectCatalogManager ().lookupTableID (caSchema, "Module");
QueryStatement qs = am.createQueryStatement (tabID);
qs.setRowIDs (new long [] rowid);
qs.setAspectFilter (am.createAspectFilter
  ("'Language' IN ('fr', 'fr_FR', 'fr_CH') OR 'Region' = 'FRA'"));

ResultRows rr = qs.execute ();
while (rr.next ()) {
  rr.getRowID ();
  AspectContextElement e = rr.getContext ().getElement ();
  for (int i = 0; i < as.count (); i++) {
    e.getIndexKeyValue (i);
  }
  for (int i = 0; i < cs.count (); i++) {
    rr.getValueString (i);
  }
}
```

Listing 6.2: Umsetzung der Anfrage `IS+*02` mittels API

Für eine effektive Realisierung der Anfrage `II1103` sind die elementaren API-Aufrufe in **Listing 6.3** dargestellt. Um die gewünschten aspektabhängigen Daten anlegen zu können, werden zunächst zur Fachtabelle MODULE ein **ModifyStatement**-Objekt erzeugt und die betreffende Produktkomponente per rowid festgelegt. Daraufhin erfolgt die Einschränkung auf die britische Sprach- und Preisregion durch einen eindeutigen Aspektkontext, wobei dem Versions-Aspekt durch fehlende Spezifikation automatisch der aktuellste Stand zugeordnet ist. Anschließend können die aspektspezifischen Daten mit Hilfe des **ColumnSet**-Objekts übergeben und letztendlich eingefügt werden. Hierbei verwendete Variablen wie langID, lang_en_GB oder descID referenzieren Idents von Aspekten, Aspektschlüsseln und Attributen aus dem Aspektkatalog. Deren Ermittlung ist nach den in Abschnitt 5.3.1 dargestellten Prinzipien vorzunehmen.

```
int tabID = am.getAspectCatalogManager ().lookupTableID (caSchema, "Module");
ModifyStatement ms = am.createModifyStatement (tabID);
ms.setRowID (rowid);

UnambiguousAspectContext uac = am.createUnambiguousAspectContext ();
AspectSet as = uac.getAspectSet ();
uac.getElement ().setIndexKeyValue (as.getIndex (langID), lang_en_GB);
uac.getElement ().setIndexKeyValue (as.getIndex (regID), reg_UK);
uac.getElement ().setIndexKeyValue (as.getIndex (pgID), pg_standard);
ms.setAspectContext (uac);

ColumnSet cs = ms.getColumnSet ();
ms.setValueString (cs.getIndex (nameID), "screw");
ms.setValueString (cs.getIndex (descID), "hex socket");
ms.setValueString (cs.getIndex (priceID), "0.33");
ms.setValueString (cs.getIndex (materialID), "stainless steel");
ms.executeInsert ();
```

Listing 6.3: Umsetzung der Anfrage `II1103` mittels API

Für eine effektive Realisierung der Anfrage `ID1*04` sind die elementaren API-Aufrufe in **Listing 6.4** dargestellt. Auch hier müssen im ersten Schritt von **ModifyStatement** eine Instanz zur Tabelle MODULE erzeugt und die relevante Produktkomponente per rowid festgelegt werden. Um darüber hinaus wie gewünscht nur aspektspezifische Preisdaten zu löschen, wird das zugehörige Attribut per *setColumnSet* angegeben. Schließlich definiert ein ebenfalls als eindeutiger Aspektkontext darstellbarer Aspektfilter die zulässige Ausprägung je abhängigem Aspekt, bevor die eigentliche Löschung durchgeführt wird.

```
int tabID = am.getAspectCatalogManager ().lookupTableID (caSchema, "Module");
ModifyStatement ms = am.createModifyStatement (tabID);
ms.setRowID (rowid);

int priceID = am.getAspectCatalogManager ().lookupColumnID (tabID, "Price");
ms.setColumnSet (am.createColumnSet (new int [] {priceID}));
ms.setAspectFilter (am.createAspectFilter
  ("'Pricegrade' = 'standard' AND 'Region' = 'EURO' AND 'Version' = '3'"));
ms.executeDelete ();
```

Listing 6.4: Umsetzung der Anfrage `ID1*04` mittels API

Für eine effektive Realisierung der Anfrage `IU1105` sind die elementaren API-Aufrufe in **Listing 6.5** dargestellt. Die darin erkennbare höhere Gesamtkomplexität gegenüber den bisherigen Anfragen liegt in folgenden drei Teilschritten begründet. Zunächst erfolgt mit Hilfe einer **QueryStatement**-Instanz zur Tabelle MODULE und dem passenden Aspektfilter die Abfrage desjenigen Datums, welches der Preisaktualisierung unterzogen werden soll. Diese Information ist zwingend vor ihrer Änderung auf ein **ResultRows**-Objekt auszulesen, da eine Versionierung des bisherigen Preises stattfinden soll. Für diesen zweiten Schritt wird nun eine **ModifyStatement**-Instanz erzeugt und mit den Aspektschlüsseln sowie den aspektspezifischen Werten des Suchergebnisses initialisiert. Anschließend erfolgt im eindeutigen Aspektkontext die Belegung des Versions-Aspekts mit der nächsten Versionsnummer (hier „15“) und daraufhin das Einfügen jener Daten als neuer Eintrag zu aspektspezifischen Daten. Im letzten Schritt wird am selben **ModifyStatement**-Objekt lediglich wieder die Zuordnung eines Wertes zum Versions-Aspekt durch die Methode *setIndexKeyNull* aufgehoben und der neue Preis für die betreffende Produktkomponente gesetzt. Mit dem so veränderten Aspektkontext findet durch *executeUpdate* die eigentliche Aktualisierung des zu Beginn ermittelten Datensatzes statt.

```
int tabID = am.getAspectCatalogManager ().lookupTableID (caSchema, "Module");
QueryStatement qs = am.createQueryStatement (tabID);
qs.setRowIDs (new long [] {rowid});
int priceID = am.getAspectCatalogManager ().lookupColumnID (tabID, "Price");
qs.setColumnSet (am.createColumnSet (new int [] {priceID}));
qs.setAspectFilter (am.createAspectFilter
  ("'Region' = 'EURO' AND 'Pricegrade' = 'standard' AND 'Version' IS NULL"));
ResultRows rr = qs.execute ();

AspectSet as = rr.getAspectSet ();
ColumnSet cs = rr.getColumnSet ();
ModifyStatement ms = am.createModifyStatement (tabID);
ms.setRowID (rowid);
ms.setColumnSet (cs);
UnambiguousAspectContext uac = am.createUnambiguousAspectContext ();
ms.setAspectContext (uac);
int versionID = am.getAspectCatalogManager ().lookupAspectID ("Version");
int versionIndex = as.getIndex (versionID);
/* copy search result into new modifystatement */
while (rr.next ()) {
  AspectContextElement e = rr.getContext ().getElement ();
  for (int i = 0; i < as.count (); i++) {
    uac.getElement ().setIndexKeyValue (i, e.getIndexKeyValue (i));
  }
  for (int i = 0; i < cs.count (); i++) {
    ms.setValueString (i, rr.getValueString (i));
  }
}
uac.getElement ().setIndexKeyValue (versionIndex, "15");
ms.executeInsert ();

uac.getElement ().setIndexKeyNull (versionIndex);
ms.setValueString (cs.getIndex (priceID), "0.44");
ms.executeUpdate ();
```

Listing 6.5: Umsetzung der Anfrage `IU1105` mittels API

Für eine effektive Realisierung der Anfrage `MS**06` sind die elementaren API-Aufrufe in **Listing 6.6** dargestellt. Analog zu Listing 6.1 wird eine **QueryStatement**-Instanz zur Tabelle MODULE erzeugt und ausgeführt. Aufgrund der fehlenden Einschränkung auf eine konkrete rowid liefert jedoch die iterative Abarbeitung des **ResultRows**-Objekts nun alle aspektspezifischen Daten als uniforme Tupel aller Produktkomponenten in MODULE.

```
int tabID = am.getAspectCatalogManager ().lookupTableID (caSchema, "Module");
QueryStatement qs = am.createQueryStatement (tabID);

ResultRows rr = qs.execute ();
while (rr.next ()) {
  rr.getRowID ();
  AspectContextElement e = rr.getContext ().getElement ();
  for (int i = 0; i < as.count (); i++) {
    e.getIndexKeyValue (i);
  }
  for (int i = 0; i < cs.count (); i++) {
    rr.getValueString (i);
  }
}
```

Listing 6.6: Umsetzung der Anfrage `MS**06` mittels API

Für eine effektive Realisierung der Anfrage `MS++07` sind die elementaren API-Aufrufe in **Listing 6.7** dargestellt. Nachdem eine **QueryStatement**-Instanz zur Tabelle MODULE erzeugt wurde, erfolgt über *setColumnSet* die Festlegung der gewünschten Attribute. Die Einschränkung des Anfrageergebnisses betrifft sowohl den Aspektfilter für Region und Preisstaffel als auch den Wertfilter für das Preis-Attribut. Schließlich liefert die Anfrage im Modus DERIVE_ONLY nur abgeleitete Ergebnistupel hinsichtlich der verschiedenen Aspektmengen für die relevanten Attribute gemäß Definition 4.12 in Abschnitt 4.2.5.

```
int tabID = am.getAspectCatalogManager ().lookupTableID (caSchema, "Module");
QueryStatement qs = am.createQueryStatement (tabID);
qs.setColumnSet (am.createColumnSet (new int [] {nameID, priceID, descID}));
qs.setAspectFilter (am.createAspectFilter
  ("'Region' IN ('EURO', 'GER', 'AUT', 'FRA', 'ITA') AND 'Pricegrade' = 'standard'"));
qs.setValueFilter ("Price >= 0.5 AND Price <= 1.0");
qs.setMode (QueryStatement.Mode.DERIVE_ONLY);

ResultRows rr = qs.execute ();
while (rr.next ()) {
  rr.getRowID ();
  AspectContextElement e = rr.getContext ().getElement ();
  for (int i = 0; i < as.count (); i++) {
    e.getIndexKeyValue (i);
  }
  for (int i = 0; i < cs.count (); i++) {
    rr.getValueString (i);
  }
}
```

Listing 6.7: Umsetzung der Anfrage `MS++07` mittels API

Für eine effektive Realisierung der Anfrage `MD**08` sind die elementaren API-Aufrufe in **Listing 6.8** dargestellt. Mit Hilfe eines **ModifyStatement**-Objekts zur Tabelle MODULE erfolgt die gewünschte Löschung aller aspektspezifischen Daten durch den Aufruf von *executeDelete*.

```
int tabID = am.getAspectCatalogManager ().lookupTableID (caSchema, "Module");
ModifyStatement ms = am.createModifyStatement (tabID);
ms.executeDelete ();
```

Listing 6.8: Umsetzung der Anfrage `MD**08` mittels API

Für eine effektive Realisierung der Anfrage `MD+*09` sind die elementaren API-Aufrufe in **Listing 6.9** dargestellt. Bis auf einen zusätzlichen Aspektfilter zur Einschränkung der zu löschenden Daten ist der Aufbau analog zur vorherigen Anfrage `MD**08`.

```
int tabID = am.getAspectCatalogManager ().lookupTableID (caSchema, "Module");
ModifyStatement ms = am.createModifyStatement (tabID);
ms.setAspectFilter (am.createAspectFilter ("'Version' IN ('1', '2')"));
ms.executeDelete ();
```

Listing 6.9: Umsetzung der Anfrage `MD+*09` mittels API

6.3.2 Generisches SQL (gSQL)

Bei Nutzung von SQL zur Realisierung der definierten Workload müssen auch die in Abschnitt 4.4.1 skizzierten Transformationen beim Zugriff auf das Persistenzmodell durchgeführt werden. Da diese Pivotisierungen jedoch unabhängig von der eigentlichen Anfrage sind, lassen sich hierfür generische Sichten definieren. Im ersten Schritt erfolgt dazu mittels der in **Listing 6.10** angegebenen Sicht CONTEXTUALVALUES die Darstellung aspektspezifischer Daten als signierte Attributwerte. Hierfür ist eine Pivotisierung nach den vier existierenden Aspekten – referenziert über ihre IDs 101, 102, 103 und 104 – notwendig.

```
CREATE VIEW ContextualValues AS
SELECT RowID, Col, AspValID AS ValID,
    COALESCE (av1.Tab, av2.Tab, av3.Tab, av4.Tab) AS Tab,
    COALESCE (av1.Value, av2.Value, av3.Value, av4.Value) AS Value,
    aa1.KeyValue AS kv1, aa2.KeyValue AS kv2, aa3.KeyValue AS kv3, aa4.KeyValue AS kv4
FROM (AspectValue av1 JOIN AspectAssign aa1 ON
                aa1.Aspect = 101 AND aa1.AspectValue = av1.AspValID)
FULL OUTER JOIN (AspectValue av2 JOIN AspectAssign aa2 ON
                aa2.Aspect = 102 AND aa2.AspectValue = av2.AspValID)
     USING (RowID, Col, AspValID)
FULL OUTER JOIN (AspectValue av3 JOIN AspectAssign aa3 ON
                aa3.Aspect = 103 AND aa3.AspectValue = av3.AspValID)
     USING (RowID, Col, AspValID)
FULL OUTER JOIN (AspectValue av4 JOIN AspectAssign aa4 ON
                aa4.Aspect = 104 AND aa4.AspectValue = av4.AspValID)
     USING (RowID, Col, AspValID)
```

Listing 6.10: SQL-View CONTEXTUALVALUES für signierte Tupel

Hauptbestandteil dieser Sicht-Definition ist für jeden Aspekt ein EQUI JOIN zwischen den Tabellen AspectValue und AspectAssign, welcher gemäß Abschnitt 5.1.3.1 und dem Ausdruck (5.10) auch im Fall der Zugriffsschicht als Aufgabe vom DBMS übernommen wird. Die Verbundergebnisse werden jeweils untereinander mittels FULL OUTER JOIN bezüglich RowID, Attribut und Attributwert verknüpft, wodurch für jeden Aspekt in der Ergebnisrelation eine separate Spalte erzeugt wird. Dabei ist der FULL OUTER JOIN zwingend erforderlich, da im Allgemeinen einerseits nicht jedes Attribut von allen funktionalen Aspekten abhängt und andererseits zu einem aspektspezifischen Wert die Zuordnung eines Aspektschlüssels fehlen kann. Letztendlich enthält die Sicht ContextualValues zu jedem aspektspezifischen Attributwert (ValID, Value) einen Datensatz sowohl mit Informationen zum fachlichen Ursprung (RowID, Col, Tab) als auch mit der zugehörigen Aspektsignatur (kv1, kv2, kv3, kv4).

Wie in Listing 6.10 zu sehen, findet für jeden FULL OUTER JOIN die Formulierung der Verbundbedingung über teilweise mit NULL belegte Attribute durch das im SQL:92-Sprachumfang enthaltene USING statt. Aufgrund der fehlenden Unterstützung dieses Konstrukts beim eingesetzten DBMS DB2 LUW 9.7 ist jedoch die semantisch äquivalente Definition der Sicht nach **Listing C.4** im Anhang notwendig. Um trotz eventuell unbelegter Vergleichsattribute einen vollständigen FULL OUTER JOIN mehrfach ausführen zu können, muss die Funktion *COALESCE* verwendet werden. Deren Parameterliste hängt allerdings direkt von der Menge der im System verfügbaren Aspekte ab.

Mit der zweiten Pivotisierung gemäß Abbildung 4.12 sollen nun basierend auf der Sicht ContextualValues vollständige signierte aufgefüllte Tupel bereitgestellt werden. Dies ist offensichtlich nur unter Kenntnis der zugehörigen Basistabelle möglich, wodurch jeweils individuelle „Zwischenschichten“ entstehen. Zur relevanten Tabelle Module im Anwendungsbeispiel zeigt **Listing 6.11** die Definition der entsprechenden Sicht V_Module. Mit deren Hilfe können sowohl die aspektspezifischen Wert der Attribute (Name, Price, Description, Norm, Material) als auch die jeweils zugeordneten Schlüsselwerte der Aspekte (Language, Region, Version, Pricegrade) in Anfragen referenziert werden.

```
CREATE VIEW V_Module AS
SELECT
    RowID,
    kv1 AS Language, kv2 AS Region, kv3 AS Version, kv4 AS Pricegrade,
    col1.Value AS Name,
    col2.Value AS Price,
    col3.Value AS Description,
    col4.Value AS Norm,
    col5.Value AS Material
FROM (SELECT * FROM ContextualValues WHERE Tab=1001 AND Col=1101)
  AS col1
FULL OUTER JOIN (SELECT * FROM ContextualValues WHERE Tab=1001 AND Col=1102)
  AS col2 USING (RowID,kv1,kv2,kv3,kv4)
FULL OUTER JOIN (SELECT * FROM ContextualValues WHERE Tab=1001 AND Col=1103)
  AS col3 USING (RowID,kv1,kv2,kv3,kv4)
FULL OUTER JOIN (SELECT * FROM ContextualValues WHERE Tab=1001 AND Col=1104)
  AS col4 USING (RowID,kv1,kv2,kv3,kv4)
FULL OUTER JOIN (SELECT * FROM ContextualValues WHERE Tab=1001 AND Col=1105)
  AS col5 USING (RowID,kv1,kv2,kv3,kv4)
```

Listing 6.11: SQL-View V_Module für aspektspezifische Produktdaten

Durch die in Listing 6.11 realisierte Verbundbedingung mittels USING ist wiederum für die Nutzung unter DB2 eine alternative Definition von V_Module über die ON-Klausel notwendig. **Listing C.5** im Anhang zeigt das zugehörige SQL-Skript. Hierbei ist jedoch zu beachten, dass nun auf beiden Seiten der Vergleichsprädikate die *COALESCE*-Funktion zum Einsatz kommen muss und darin auftretende NULL-Werte in den Platzhalter −1 umzuwandeln sind. Dies ist erforderlich, um Prädikate der Form (NULL=NULL) in die Aussage (−1=−1) zu überführen, wodurch sie ebenfalls von SQL als erfüllt interpretiert werden. Diese Prädikate entsprechen der Bedingung (4.14) in Definition 4.11, die eben auch bei Belegung mit ⊥ für gleiche Aspekte wahr ist. Dadurch ist die Vereinbarkeit zweier Tupel aus ContextualValues hier bereits gewährleistet, da die vom FULL OUTER JOIN verbundenen Relationen bezüglich der fachlichen Attribute (Col=x) disjunkt sind.

Während V_Module signierte und hinsichtlich des Relationenschemas von Module aufgefüllte Tupel bereitstellt, ergibt sich im Fall von Filterbedingungen über nicht-uniforme Tupel der Zugriff auf abgeleitete Tupel wie in Definition 4.12 eingeführt. Beispielsweise ist in `MS++07` eine wertspezifische Bedingung für den Preis formuliert, wobei jenes Attribut von anderen Aspekten beeinflusst wird als die von der Anfrage ebenso gewählten Attribute Name und Description. Für solche Fälle bietet die Sicht V_Module_Derived die Menge der abgeleiteten Tupel zur Tabelle Module, **Listing C.6** im Anhang zeigt das zugehörige SQL-Skript. Gegenüber V_Module sind große strukturelle Gemeinsamkeiten erkennbar, insbesondere gilt auch hier die Disjunktheit der fachlichen Attribute in jeweils zwei mit FULL OUTER JOIN verknüpften Relationen. Gemäß Definition 4.12 genügt deshalb die Prüfung der Bedingungen (4.20) und (4.21). Hierzu kommt unabhängig vom DBMS die *COALESCE*-Funktion auf beiden Seiten des Vergleichs zum Einsatz, wobei die Ausprägungen des jeweils betrachteten Aspekts in unterschiedlichen Reihenfolgen übergeben werden. Liefern dennoch beide Funktionsaufrufe das gleiche Ergebnis, sind die Kriterien zur Ableitbarkeit bezüglich des Aspekts erfüllt.

Unter Beachtung von DBMS-spezifischen Besonderheiten können die drei vorgestellten Sichten ContextualValues, V_Module und V_Module_Derived bezüglich einer Menge von Aspekten automatisch für jede relevante Fachtabelle erzeugt werden. Insbesondere liegt keine Abhängigkeit zu konkreten Workloads vor. Deren Umsetzung mit SQL basierend auf jenen Hilfssichten ist in **Anhang C.2** dargestellt. Dabei dient `$r$` als Platzhalter für die RowID einer spezifischen Produktkomponente. Zudem müssen Aspektschlüsselwerte über ihre zuvor ermittelten Idents im Aspektkatalog angegeben werden.

6.3.3 Spezifisches SQL (sSQL)

Auch beim dritten Testkandidaten sSQL erfolgt die Umsetzung der Workload ausschließlich auf Basis von SQL. Allerdings wird im Gegensatz zum gSQL-Ansatz auf jegliche generische Sichten verzichtet. Die notwendigen Pivotisierungen besitzen zwar die gleichen Verbund-Grundstrukturen wie in Abschnitt 6.3.2 erläutert, allerdings beinhalten diese nur die für eine Anfrage tatsächlich notwendigen Elemente. Zudem wurde durch manuelle Optimierung eine möglichst effiziente Formulierung der jeweiligen Verbund-Bedingungen gesucht. Obwohl diese Aufgabe eigentlich vom Optimizer im DBMS automatisch durchgeführt wird, ergaben sich im Test für semantisch äquivalente Anfragen stark abweichende Antwortzeiten. Die letztlich für den Performancetest verwendeten SQL-Skripte sind im **Anhang C.3** zu finden, wobei für die Anfragen `II1103`, `MD**08` und `MD+*09` das jeweilige SQL-Skript des generischen Ansatzes übernommen wurde.

6.4 Durchführung

Nachdem im bisherigen Verlauf von Kapitel 6 sowohl die technischen als auch die inhaltlichen Rahmenbedingungen für den Performancetest definiert wurden, lässt sich nun dessen eigentliche Ausführung betrachten. Dazu findet in Abschnitt 6.4.1 die Beschreibung des Testablaufs statt. Anschließend erfolgt die Präsentation der Antwortzeiten aller Anfragen in Abschnitt 6.4.2.

6.4.1 Vorgehen

Die Abarbeitung eines einzelnen Testlaufs ist in **Abbildung 6.3** dargestellt. Wie auf der linken Seite zu erkennen, ist ein solcher Durchlauf abhängig vom DBMS, der konkreten Workload und dem Testkandidaten wie in den vorherigen Abschnitten erläutert. Darüber hinaus stehen noch die drei in Tabelle 6.1 spezifizierten Datenbestände zur Verfügung. Aufgrund der Unabhängigkeit jener vier Dimensionen ergibt sich kombinatorisch eine größere Menge an verschiedenen Testläufen. Deren Ausführung wurde mit Hilfe eines zusätzlichen in [Pie11b] näher beschriebenen Tools weitgehend automatisiert.

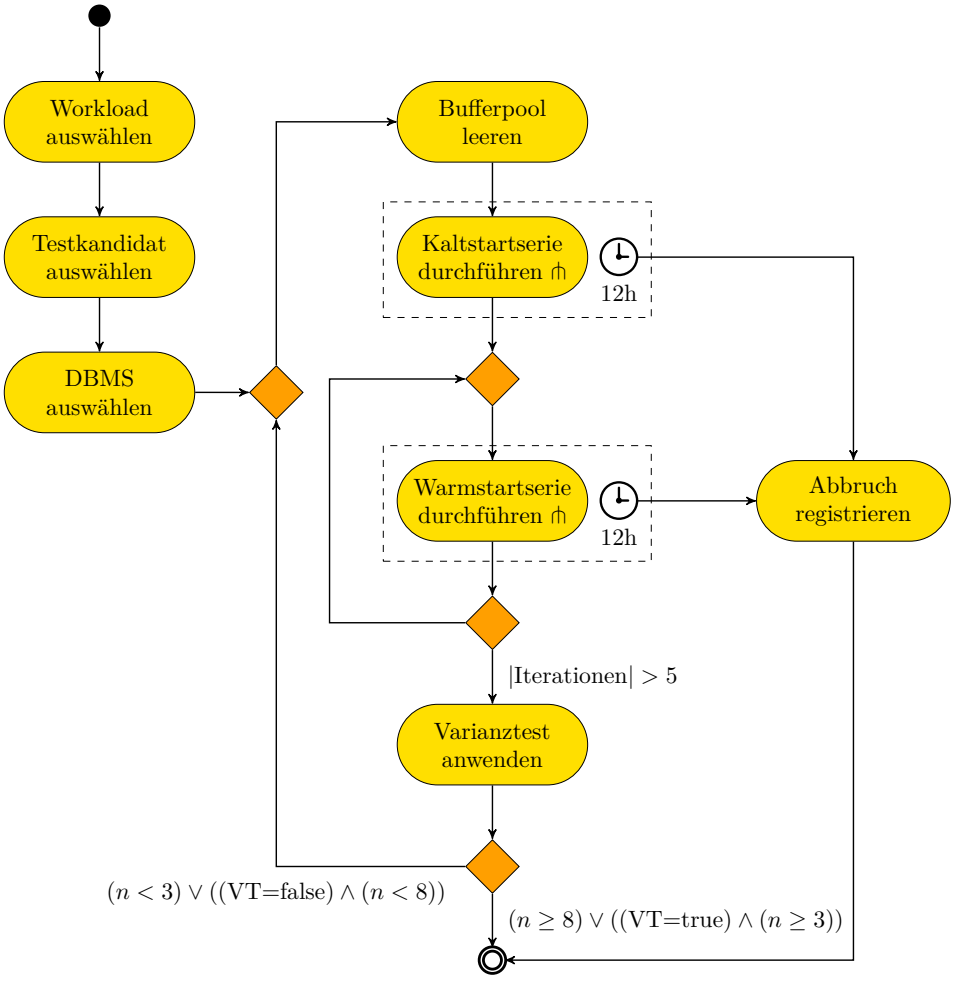


Abbildung 6.3: UML-Aktivitätsdiagramm für den Testablauf nach [Pie11b]

Zur Vorbereitung der Testläufe werden zunächst die Testdatensätze für die Datenbestände DS01, DS02 und DS03 mit dem Tool **RandomFillUniform** erzeugt. Nachdem gemäß der Aufrufparameter die Anfrage, der Testkandidat und das DBMS festgelegt sind, erfolgt durch Neustart des DBMS die Leerung des Bufferpools. Im Rahmen der anschließenden sogenannten Kaltstartserie wird die Anfrage nacheinander gegen alle drei Datenbestände gestellt. Ist dies abgeschlossen, wird die gleiche Anfrage erneut innerhalb einer Warmstartserie, d.h. ohne den Bufferpool zu leeren, gegen jene drei Datenbestände gestellt. Nach Ende der Warmstartserie wird diese noch weitere fünf Mal ausgeführt, um aufgrund starker Schwankungen bei den Antwortzeiten für eine Warmstartphase einen aussagekräftigen Durchschnittswert bilden zu können. Gibt eine Anfrage innerhalb einer Testserie hinsichtlich eines Datenbestands nach 12 Stunden noch kein Ergebnis zurück, wird sie abgebrochen und für weitere Durchläufe (im Fall der Warmstartserie) ausgeschlossen.

Nach Abschluss aller sieben Testserien (einmal Kaltstart, sechsmal Warmstart) erfolgt ein Varianztest (VT). Hierbei werden die von den Testläufen in einer Datenbank hinterlegten Antwortzeiten bezüglich der aktuell geltenden Aufrufparameter sowie getrennt für Kalt- und Warmstartserie auf starke Schwankungen getestet. Ein solcher Ausreißer soll vorliegen, wenn die mittlere Abweichung dieser Werte größer als ein Fünftel[1] des Durchschnittswerts jener Werte ist. In diesem Fall ist der VT nicht erfüllt und der komplette Testlauf wird gemäß der linken Kante in Abbildung 6.3 erneut ausgeführt.

Ist der VT sowohl für die Antwortzeiten aus der Kaltstartserie und der Warmstartserie als auch hinsichtlich aller Datenbestände erfüllt, wird der Testlauf unter den Startbedingungen erfolgreich beendet. Dadurch können die hinterlegten mittleren Laufzeiten als zuverlässig betrachtet werden. Hierfür muss die Anzahl an Gesamtdurchläufen, bezeichnet mit n im Flussdiagramm, jedoch zwischen einer unteren und oberen Grenze liegen. Somit muss selbst bei einem erfolgreichen VT der Testlauf mindestens dreimal ausgeführt werden. Scheitert dieser jedoch selbst nach achtmaliger Ausführung am VT, wird auf einen weiteren Testlauf verzichtet und dies in den Ergebnissen entsprechend vermerkt.

6.4.2 Ergebnisse

Gemäß der im vorherigen Abschnitt 6.4.1 erläuterten Parametrisierung für einen Testlauf lässt sich die Menge aller Antwortzeiten bezüglich der vier Dimensionen Datenbestand, Workload, Testkandidat sowie DBMS klassifizieren. Wie in **Tabelle 6.2** dargestellt, werden die Ergebnisse primär nach den beiden betrachteten Datenbanksystemen PostgreSQL und DB2 unterschieden. Dabei findet innerhalb des DBMS eine Differenzierung hinsichtlich der Testkandidaten gSQL, sSQL und API statt. Mit der analogen Vorgehensweise für die zwei verbleibenden Parameter enthält schließlich jede Tabellenzeile die Antwortzeiten zu einer konkreten Anfrage und einem der drei Datenbestände. Aufgrund der jeweils identischen SQL-Realisierung der Workloads `II1103`, `MD**08` und `MD+*09` durch die Testkandidaten gSQL und sSQL wird in diesen Fällen das Ergebnis nur einmal zentriert dargestellt.

Darüber hinaus kann eine Zelle durch verschiedene Formatierungen zusätzliche Informationen widerspiegeln. Auffälligstes Merkmal hierbei ist der Rahmen um die kürzeste Antwortzeit je DBMS. Im Fall kursiver Werte ist dagegen eine vorsichtigere Bewertung notwendig,

[1] Dieser Grenzwert wurde willkürlich festgelegt.

		PostgreSQL			DB2		
Anfrage	Daten	gSQL	sSQL	API	gSQL	sSQL	API
IS**01	DS01	1.157	*110*	30	32.702	507	24
	DS02	9.540	*93*	26	1.297.542	696	335
	DS03	90.328	148	55	>43.200.000	688	52
IS+*02	DS01	1.020	110	40	31.861	558	34
	DS02	8.972	84	34	1.292.059	670	307
	DS03	80.867	84	74	>43.200.000	552	53
II1103	DS01	*390*		*124*	*699*		*295*
	DS02	*365*		*64*	*262*		*188*
	DS03	*197*		*164*	*264*		*176*
ID1*04	DS01	333	50	*233*	17.233	237	196
	DS02	1.705	459	307	681.569	695	90
	DS03	19.708	419	348	32.661.365	325	163
IU1105	DS01	916	*325*	51	18.534	*240*	161
	DS02	2.428	*188*	52	701.006	*880*	272
	DS03	21.594	1.288	188	32.824.311	2.161	143
MS**06	DS01	1.276	876	338	31.344	10.113	388
	DS02	9.628	2.251	1.235	1.284.312	288.380	1.075
	DS03	81.443	41.137	3.626	>43.200.000	16.091.451	5.762
MS++07	DS01	err.	err.	442	30.642	5.997	626
	DS02	err.	err.	1.581	1.313.547	140.342	1.587
	DS03	err.	err.	4.036	>43.200.000	6.741.075	8.345
MD**08	DS01	382		365	760		683
	DS02	909		865	4.132		3.828
	DS03	9.719		10.611	28.186		28.244
MD+*09	DS01	39		328	313		441
	DS02	37		975	214		1.184
	DS03	171		5.300	297		9.849

Tabelle 6.2: Ergebnisse der Testdurchführung (Antwortzeiten in ms)

da hiermit ein fehlgeschlagener Varianztest gemäß der Definition in Abschnitt 6.4.1 gekennzeichnet wird. Lieferte dagegen eine spezifische Anfrage-Konstellation im Zeitlimit von 12h überhaupt kein Ergebnis, ist dies mit „>43.200.000“ hinterlegt. Schließlich wurde die Anfrage `MS++07` unter PostgreSQL sofort abgebrochen mit Hinweis auf eine nicht unterstützte Verbundbedingung[2]. Hierfür stehen die Werte „err.“ in der Tabelle.

[2]Fehlermeldung: `FULL JOIN is only supported with merge-joinable join conditions`.

6.5 Auswertung

Ziel dieses letzten Teils von Kapitel 6 ist die Bewertung der in Tabelle 6.2 präsentierten Testergebnisse. Dabei soll die grafische Aufbereitung von jeweils relevanten Daten die Analyse unterstützen. Aufgrund der um Größenordnungen langsameren Antwortzeiten gegenüber den beiden anderen Testkandidaten findet zunächst eine separate Betrachtung des gSQL-Ansatzes in Abschnitt 6.5.1 statt. Daraufhin wird in Abschnitt 6.5.2 die API-Zugriffsschicht mit dem sSQL-Ansatz verglichen. Eine Zusammenfassung wichtiger Erkenntnisse erfolgt abschließend in Abschnitt 6.5.3.

6.5.1 Verhalten des gSQL-Ansatzes

Die Auswertung des Laufzeitverhaltens unter Verwendung von gSQL erfolgt getrennt nach ausschließlich lesenden sowie modifizierenden Anfragen.

6.5.1.1 Lesende Anfragen

Durch Fokussierung auf nur einen Testkandidaten ist ein direkter Vergleich der beiden Datenbanksysteme naheliegend. Exemplarisch für den Datenbestand DS01 veranschaulicht **Abbildung 6.4** jene Gegenüberstellung aller lesenden Anfragen `IS**01`, `IS+*02`, `MS**06` und `MS++07`. Dabei sei auf den fehlenden Wert im Fall von `MS++07` für PostgreSQL und die bereits in Abschnitt 6.4.2 erwähnten Hintergründe hingewiesen.

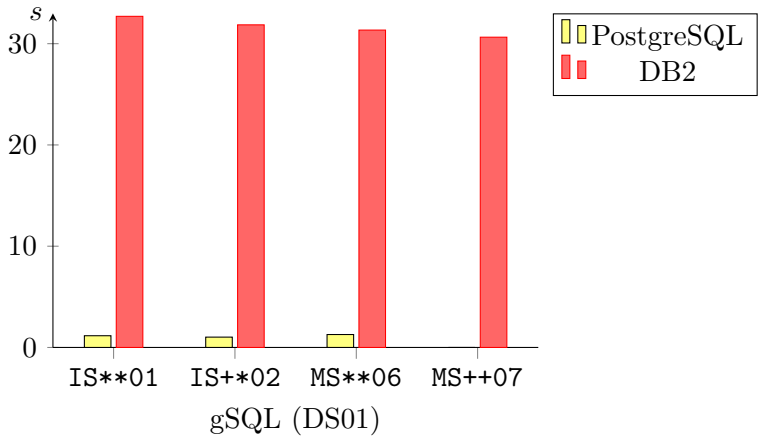


Abbildung 6.4: Antwortzeiten für lesende Anfragen bei gSQL

Auffällig sind auf den ersten Blick die enormen Differenzen von über 30 Sekunden zwischen DB2 und PostgreSQL. Eine Ursache hierfür ist offensichtlich die abweichende Definition der zugrunde liegenden Sichten V_Module und ContextualValues für DB2 aufgrund des nicht unterstützten USING in der Verbundbedingung. Dieser Abstand steigt bereits beim Datenbestand DS02 auf über 20 Minuten an, für die dritte Datenmenge kann DB2 innerhalb von 12 Stunden kein Ergebnis mehr liefern. Allerdings liegen bei PostgreSQL in diesem Fall die Antwortzeiten zur Abfrage aspektspezifischer Daten *eines* fachlichen Datensatzes auch schon im inakzeptablen Minutenbereich.

Darüber hinaus zeigt sich, dass die Antwortzeit zwar sehr stark vom DBMS und dem Datenbestand beeinflusst wird, die eigentliche Workload jedoch fast keine Auswirkung hat. Dieser Effekt ist in Abbildung 6.4 insbesondere für DB2 gut zu erkennen und leitet sich auch für die anderen Datenmengen aus Tabelle 6.2 ab. Dies ist insofern ein überraschendes Ergebnis, da die Anfragen `IS**01` und `IS+*02` effektiv nur ein einziges fachliches Tupel betreffen. Trotzdem entsprechen die zugehörigen Antwortzeiten ziemlich exakt denjenigen für die Massenanfragen `MS**06` und `MS++07`. Daraus lässt sich schließen, dass beide DBMS die notwendigen Transformationen *immer* für den gesamten Datenbestand durchführen und zusätzliche Filter-Bedingungen erst danach anwenden. Ein solches Vorgehen sollte jedoch eigentlich vom internen Optimizer vermieden werden und stellt effektiv einen Fehler an dieser Komponente in den DBMS-Produkten dar.

6.5.1.2 Modifizierende Anfragen

Wie in Abschnitt 6.3.3 erläutert und anhand der Ergebnisse in Tabelle 6.2 zu sehen, wurden die Anfragen `II1103`, `MD**08` und `MD+*09` jeweils mit identischen SQL-Anweisungen für die Testkandidaten gSQL und sSQL umgesetzt. Deren Auswertung findet deswegen im Rahmen von Abschnitt 6.5.2 statt. Die Antwortzeiten der verbleibenden modifizierenden Workloads `ID1*04` und `IU1105` sind für alle drei Datenbestände in **Abbildung 6.5** paarweise für DB2 und PostgreSQL gegenübergestellt.

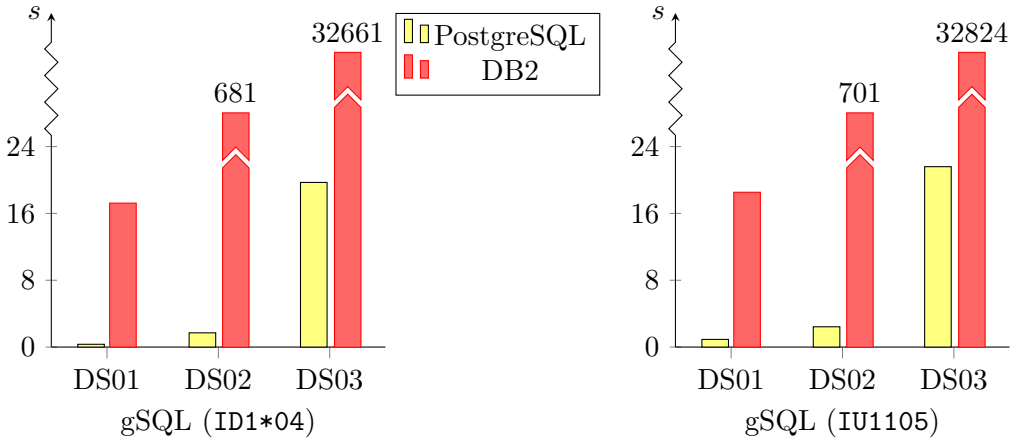


Abbildung 6.5: Antwortzeiten für `ID1*04` und `IU1105` bei gSQL

Dabei zeigen beide Anfragen trotz unterschiedlicher inhaltlicher Ausrichtung ein fast identisches Verhalten bei allen Testsituationen. Die Ursache hierfür ist durch genauere Betrachtung der zugrunde liegenden SQL-Anweisungen in **Listing C.10** und **Listing C.11** im Anhang C.2 zu finden. In beiden Fällen wird zur Bestimmung des relevanten Tupels die Sicht ContextualValues verwendet. Damit hat deren zuvor erläutertes zeitliches Verhalten natürlich direkten Einfluss auf die Antwortzeit der gesamten Anfrage. Während PostgreSQL selbst für die größte Datenmenge noch ein Ergebnis im zweistelligen Sekundenbereich liefert, ist diese Grenze von DB2 nur für die kleinste Datenmenge erreichbar. Die Reaktionszeiten der zwei weiteren Datenbestände liegen mit über 10 Minuten (DS02) und neun Stunden (DS03) weit entfernt von jeglichen Akzeptanzkriterien zur Modifikation nur *eines einzigen* fachlichen Datensatzes.

6.5.2 Zugriffsschicht versus sSQL-Ansatz

Analog zum Abschnitt 6.5.1 erfolgt der Vergleich von Antwortzeiten der Zugriffsschicht und des sSQL-Ansatzes getrennt für lesende und ändernde Anfragen. Dabei werden je Workload die Zeiten in beiden Datenbanksystemen für alle drei Datenbestände gegenübergestellt.

6.5.2.1 Lesende Anfragen

Die inhaltliche Ähnlichkeit der Individualanfragen `IS**01` und `IS+*02` setzt sich erwartungsgemäß in den Ergebnissen fort, was die Gegenüberstellung von **Abbildung 6.6** und **Abbildung 6.7** sehr deutlich zeigt. Offensichtlich führt die zusätzliche Filterbedingung in `IS+*02` zu keiner nennenswerten Veränderung. Dagegen liegen die Antwortzeiten der API-Zugriffsschicht in jedem Fall unter denen des sSQL-Ansatzes, wobei es sich nur um Differenzen von Zehntelsekunden handelt. Bis auf den unerklärbaren Ausreißer bei DS02 unter DB2 spielt zudem das DBMS für die API keine entscheidende Rolle.

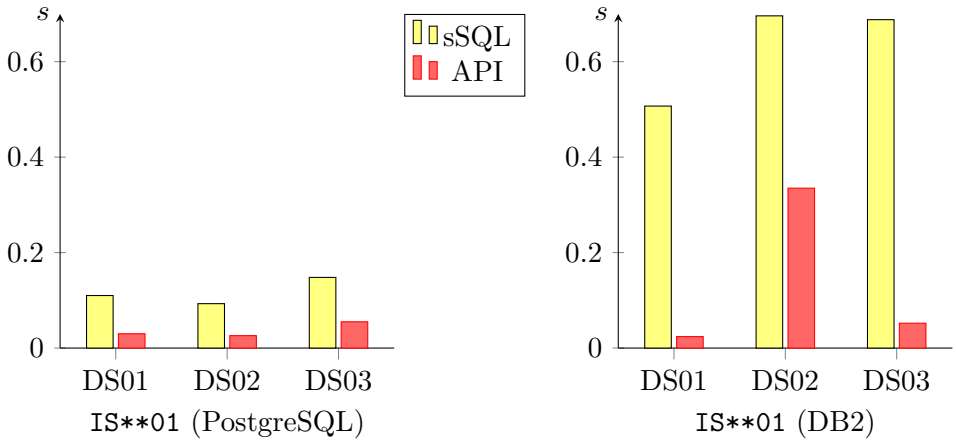


Abbildung 6.6: Antwortzeiten für `IS**01` bei sSQL und API

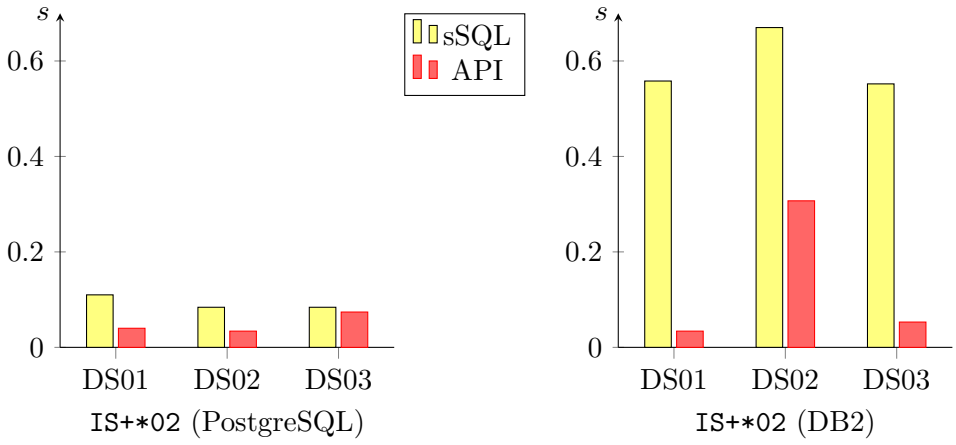


Abbildung 6.7: Antwortzeiten für `IS+*02` bei sSQL und API

Allerdings benötigt der sSQL-Testkandidat unter DB2 sehr viel mehr Zeit als unter PostgreSQL. Wie beim gSQL-Ansatz wird auch hier der Rückgriff auf *COALESCE* bei DB2 als Ersatz für das fehlende USING in der Verbund-Bedingung die Ursache sein.

Ein vollkommen anderes Verhalten zeigt **Abbildung 6.8** für die Massenanfrage `MS**06`. Während sich die Beantwortung der Workload durch manuell optimiertes SQL insbesondere bei DB2 mit steigendem Datenbestand extrem verzögert, bleibt der API-Ansatz im Vergleich dazu fast konstant. Zwar steigen auch hier die Antwortzeiten auf annähernd sechs Sekunden bei DS03, fallen aber um Größenordnungen geringer aus als bei sSQL.

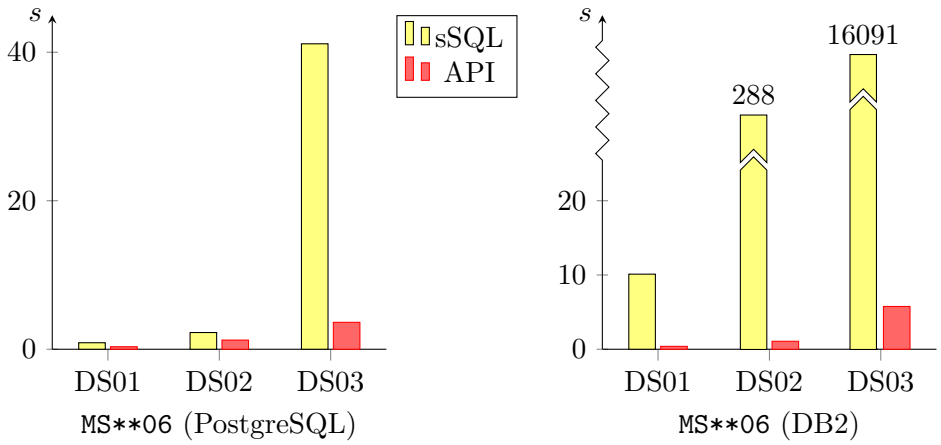


Abbildung 6.8: Antwortzeiten für `MS**06` bei sSQL und API

Abgesehen von den fehlenden Werten unter PostgreSQL entsprechend der Hinweise in Abschnitt 6.4.2 präsentiert sich die Auswertung von `MS++07` in **Abbildung 6.9** analog zu `MS**06`. Aufgrund der zusätzlichen Filterbedingung sollten sich jedoch kürzere Antwortzeiten für geringere Datenmenge ergeben. Während dies im Fall des sSQL-Ansatzes durch geschickte Integration des Aspektfilters gelungen ist, zeigt die API einen geringen gegenteiligen Effekt. Hier scheint der Aufwand zur Verarbeitung jener Filterbedingungen höher zu sein als die Zeit zur Prozessierung der dadurch eingesparten Daten.

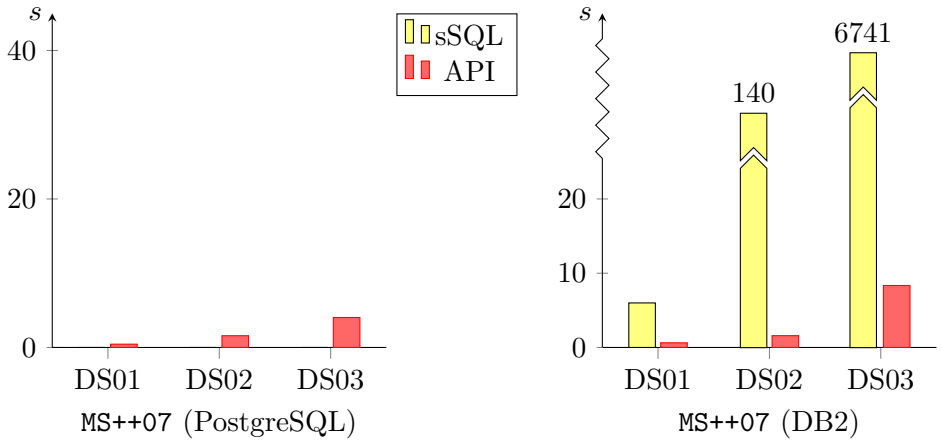


Abbildung 6.9: Antwortzeiten für `MS++07` bei sSQL und API

6.5.2.2 Modifizierende Anfragen

Das Laufzeitverhalten von Anfrage `II1103` unter den verschiedenen Bedingungen wird in **Abbildung 6.10** dargestellt. Dabei ist zu beachten, dass gemäß der Formatierung in Tabelle 6.2 jegliche Antwortzeiten sowohl beim sSQL-Ansatz als auch für die API unter einem fehlgeschlagenen Varianztest entstanden sind. Die damit verbundene hohe Schwankungsbreite der einzelnen Messwerte erlaubt im direkten Vergleich keine weiteren Schlussfolgerungen, obwohl die Zugriffsschicht gegenüber sSQL tendenziell schneller ist.

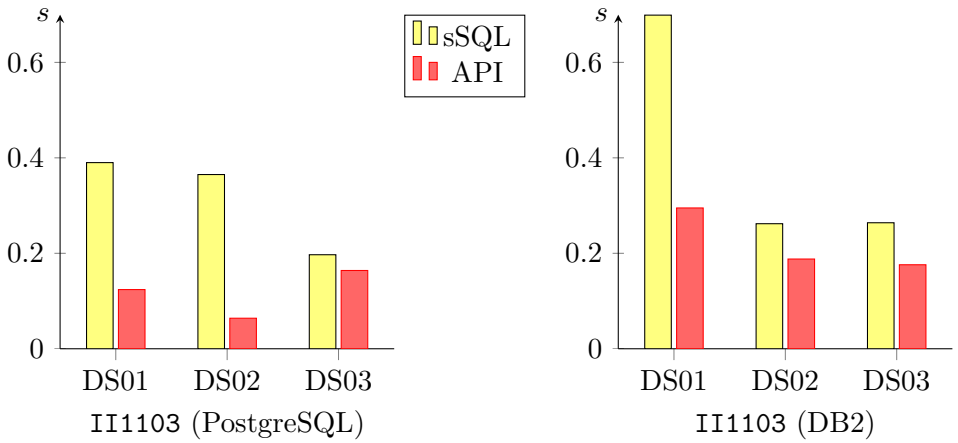


Abbildung 6.10: Antwortzeiten für `II1103` bei sSQL und API

Die Ergebnisse der Workload `ID1*04` konnten den Varianztest zwar bis auf eine Testbedingung erfüllen, jedoch zeigt **Abbildung 6.11** ein ähnlich sprunghaftes Verhalten wie bei `II1103`. Insbesondere lässt sich aus wachsenden Datenbeständen keine Laufzeitverschlechterung ableiten wie bei den lesenden Anfragen in Abschnitt 6.5.2.1. Darüber hinaus ist abweichend vom bisherigen Trend PostgreSQL nicht durchgehend das schnellere DBMS, was sogar für beide Testkandidaten gilt. Allerdings bestätigt sich auch hier wieder die höhere Leistungsfähigkeit der API verglichen mit dem manuell optimierten SQL.

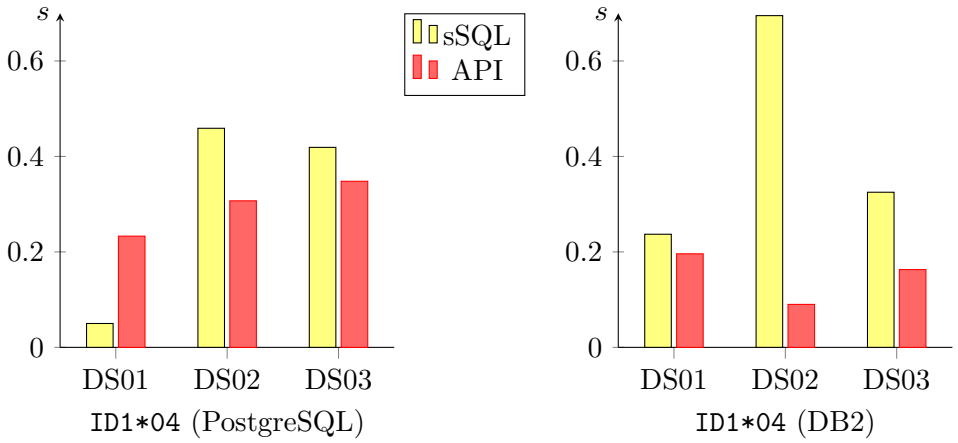


Abbildung 6.11: Antwortzeiten für `ID1*04` bei sSQL und API

Im Gegensatz zu den beiden vorherigen Workloads sind für die Anfrage `IU1105` zwischen den beiden Testkandidaten in **Abbildung 6.12** signifikante Unterschiede erkennbar. Auf der einen Seite kann die Anfrage von der API sehr viel schneller bearbeitet werden als vom sSQL-Ansatz. Andererseits reagiert letzterer empfindlich auf größere Datenbestände unabhängig vom DBMS, während die Antwortzeiten der Zugriffsschicht diesbezüglich nur marginal steigen beziehungsweise im Fall von DB2 gar nicht negativ beeinflusst werden.

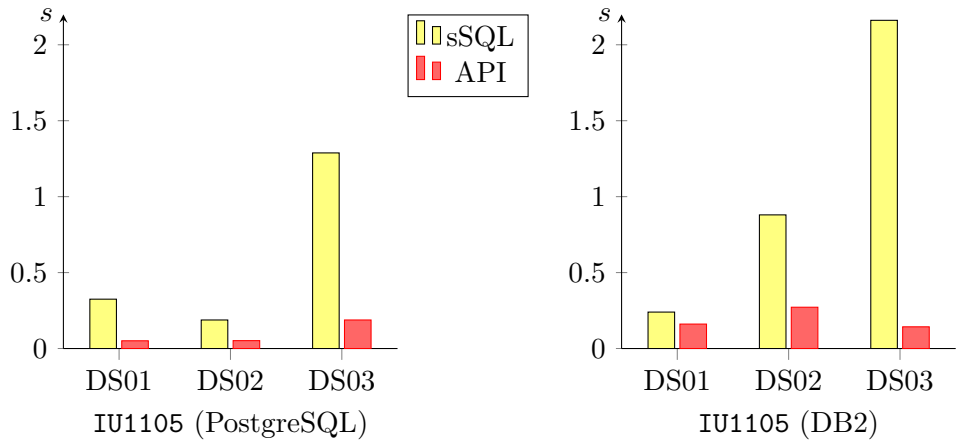


Abbildung 6.12: Antwortzeiten für `IU1105` bei sSQL und API

Nach den selektiven Änderungen zeigt nun **Abbildung 6.13** das Verhalten für die Massen-Löschung durch Workload `MD**08`. Dabei müssen gemäß Tabelle 6.1 beispielsweise für DS03 knapp 60.000 Datensätze mit aspektspezifischen Informationen zur Tabelle MODULE entfernt werden. Hierfür benötigen sowohl PostgreSQL als auch DB2 einige Sekunden, wobei sich die Antwortzeiten beider DBMS um den Faktor drei unterscheiden. Dagegen stimmen jeweils die Werte des sSQL-Ansatzes und der API bis auf wenige Millisekunden überein. Die Ursache hierfür liegt in der Zugriffsschicht-Komponente zur Vorverarbeitung von Filterbedingungen. Diese erkennt, dass eine Transformation der Daten nicht notwendig ist und generiert eine zum sSQL-Ansatz semantisch äquivalente Anfrage.

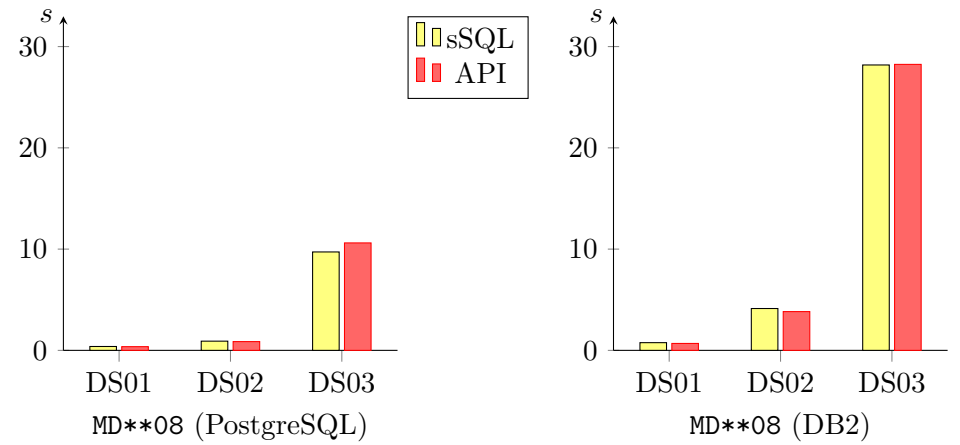


Abbildung 6.13: Antwortzeiten für `MD**08` bei sSQL und API

Abschließend seien noch die Ergebnisse in **Abbildung 6.14** für die letzte Anfrage `MD+*09` betrachtet. Im Gegensatz zum bisherigen Verhalten der Zugriffsschicht stellt sich hier verglichen mit dem direkten SQL-Ansatz eine vollkommen andere Situation dar. Einerseits sind die Antwortzeiten der API teilweise um Größenordnungen langsamer. Andererseits hängen diese offensichtlich vom zugrunde gelegten Datenbestand ab, was bei sSQL nicht erkennbar ist.

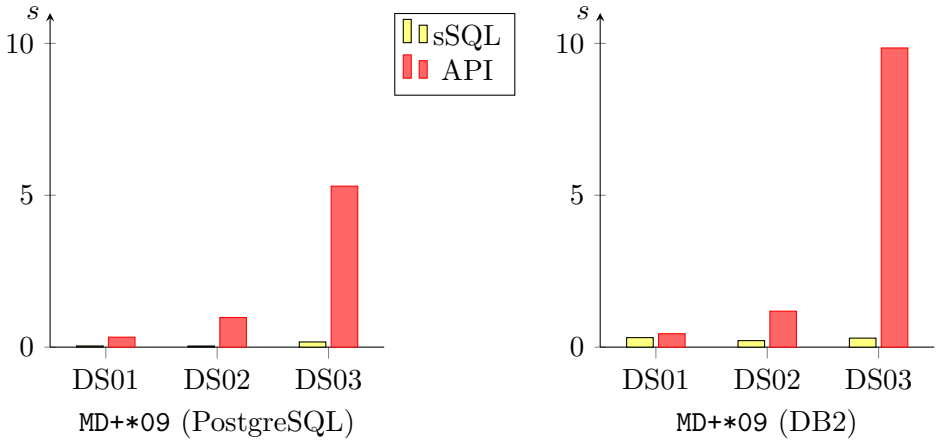


Abbildung 6.14: Antwortzeiten für `MD+*09` bei sSQL und API

Die Ursache für beide Phänomene liegt wiederum in der für Filter-Analysen zuständigen Zugriffsschicht-Komponente. Aufgrund der einzigen Einschränkung auf Schlüsselwerte des Versions-Aspekts kann für die Anfrage `MD+*09` die aufwändige Transformation der Daten entfallen. Während das im manuellen SQL-Skript in **Listing C.24** berücksichtigt wurde, führt die API diesen Schritt zunächst aus, bevor die zu löschenden aspektspezifischen Werte ermittelt werden. Dadurch sind auch umfangreiche Datenmengen an Attributwerten sowie zu löschenden IDs zwischen DBMS und der Zugriffsschicht zu transportieren. Hier zeigt sich also noch Potential zur Erweiterung der prototypischen Implementierung, um derartige Situationen auch in der API identifizieren und optimieren zu können.

6.5.3 Fazit

Im Vergleich der drei Testkandidaten aus Abschnitt 6.3 zeigt die API-Zugriffsschicht das mit Abstand beste Antwortverhalten für die getestete Workload. Abgesehen von `MD+*09` und einzelnen Datenbeständen bei `ID1*04` und `MD**08` konnte die Zugriffsschicht die Anfragen am schnellsten beantworten. Dagegen waren die Ansätze gSQL und sSQL mitunter um einige Größenordnungen langsamer abhängig vom verwendeten DBMS, welches für die Zugriffsschicht bis auf wenige Ausnahmen keine signifikante Rolle spielt. Die Einordnung jener Testergebnisse im Kontext der gesamten Arbeit wird nun im folgenden Kapitel 7 zusammenfassend diskutiert. Darüber hinaus kann dem API-Ansatz in Verbindung mit dem Referenzmodell für die aspektorientierte Datenhaltung die effektive Praxistauglichkeit bescheinigt werden.

Kapitel 7

Zusammenfassung und Ausblick

Dieses Kapitel schließt die vorliegende Arbeit ab. Dazu werden einerseits die erreichten Ergebnisse in **Abschnitt 7.1** resümiert und andererseits in **Abschnitt 7.2** weiterführende Themen aufgezeigt.

7.1 Ergebnisse

Nachfolgend findet eine Zusammenfassung der behandelten Themengebiete gemäß ihrer chronologischen Abfolge über die einzelnen Kapitel statt.

Produktkonfiguratoren (Kapitel 2)

Motiviert durch den einleitend identifizierten Trend der Individualisierung im Kontext des Konsumverhaltens wurde zur Bewältigung der damit verbundenen Herausforderungen innerhalb von Kapitel 2 die Anwendungsklasse der Produktkonfiguratoren detailliert betrachtet. Hierbei fand neben grundlegenden Begriffsprägungen und Definitionen auch die Analyse notwendiger Voraussetzungen statt, wie beispielsweise die Produkt-Modularisierbarkeit. Daraufhin erfolgte die Vorstellung einer Klassifikations-Systematik potentieller Konfigurationssysteme hinsichtlich betriebswirtschaftlicher, technologischer und organisatorischer Dimensionen als zentrales Thema des Kapitels. Basierend auf jener Systematik konnte ein generalisiertes integratives Grundmodell für Produktkonfiguratoren unter Nutzung des MVC-Patterns entworfen werden. Abschließend wurden anhand der Mehrsprachigkeit als Teil der Globalisierung einige Auswirkungen auf die Datenhaltung in Produktkonfiguratoren betrachtet.

Aspektorientierte Datenhaltung (Kapitel 3)

Ausgehend von den zuvor analysierten Konsequenzen der Globalisierung auf die Datenhaltung in Produktkonfiguratoren diente das Kapitel 3 einer verallgemeinerten Betrachtung jener Herausforderungen. Dazu erfolgte zunächst die Charakterisierung und Definition der hierbei relevanten querschnittlichen Belange als funktionale Aspekte. Für deren Beherrschung wurde daraufhin das Paradigma der Aspektorientierten Datenhaltung (AOD)

inklusive fünf zentraler Anforderungen formuliert. Abschließend fand eine Analyse und Bewertung der Persistierungsmodelle RDBMS, ORDBMS und XML hinsichtlich ihrer Nutzbarkeit für das AOD-Paradigma statt. Hierbei ging das relationale DBMS im Vergleich und insbesondere aufgrund der größten Praxisrelevanz eindeutig als Favorit hervor. Darüber hinaus wurde das Anwendungsbeispiel MODULE vorgestellt, welches die Verwaltung von Produktkomponenten unter Einfluss funktionaler Aspekte simuliert und im Verlauf der Arbeit durchgängig für die Veranschaulichung abstrakter Sachverhalte zum Einsatz kam.

Relationales Referenzmodell (Kapitel 4)

Nach der vorangegangenen Festlegung auf RDBMS als geeignetes Datenmodell für die praxistaugliche Unterstützung des AOD-Paradigmas widmete sich Kapitel 4 dem Entwurf eines zugehörigen Referenzmodells. Dazu fand im ersten Teil durch die Definition von Begriffen wie Aspektsignatur und Aspektkontext eine Formalisierung aspektspezifischer Daten statt, welche im Folgenden die Grundlage einer klaren und eindeutigen Spezifikation bildete. Anschließend wurden die konkreten Tabellendefinition des Persistenzmodells unter Verwendung des EAV-Konzepts vorgestellt und deren prinzipielle Funktionsweise anhand des Anwendungsbeispiels demonstriert. Neben den reinen Speicherstrukturen stand auch das Zugriffsmodell inklusive einer zweistufigen Pivotisierung bezüglich der Attributwerte und Aspektsignaturen im Fokus der Betrachtungen. Im letzten Teil wurden schließlich verschiedene Technologien zur Realisierung eines solchen Zugriffsmodells analysiert und auf Basis wichtiger Kriterien wie Praxisrelevanz und Performance verglichen. Dabei zeigte sich, dass der Einsatz einer zusätzlichen Zugriffsschicht zwischen Anwendungsprogramm und DBMS zur Realisierung der notwendigen Pivotisierungen gegenüber der Formulierung mit reinen SQL-Mitteln zu präferieren ist.

Prototypische Implementierung (Kapitel 5)

Mit Kapitel 5 wurde der Nachweis zur Realisierbarkeit der favorisierten Zugriffsschicht in Form einer prototypischen Implementierung erbracht. Hierzu erfolgten zunächst die Analyse und Festlegung möglicher Freiheitsgrade bei der Ausgestaltung einer solchen Zugriffsschicht. In diesem Zusammenhang betrachtete Kriterien waren unter anderem der Kopplungsgrad zum DBMS, die Fähigkeit zur Zwischenspeicherung sowie die Aufgabenverteilung bei der Datentransformation zwischen Persistierungs- und Zugriffsschicht. Weiterhin fand die Formalisierung und Definition zulässiger Bedingungen für Aspekt- und Wertfilter in Form von Syntax-Diagrammen statt. Daraufhin wurden im Hauptteil geeignete Datenstrukturen und Methoden-Signaturen sowie wichtige Implementierungs-Details einer API für die Zugriffsschicht in Java spezifiziert, mit denen sich sowohl die Metadaten im Aspektkatalog als auch die aspektspezifischen Daten selbst verwalten lassen. Die prinzipielle Verwendbarkeit jener API konnte abschließend durch Code-Fragmente mit Bezug zum zentralen Anwendungsbeispiel demonstriert werden.

Performancetest (Kapitel 6)

Das letzte Kapitel 6 umfasste die Analyse der Praxistauglichkeit für die Zugriffsschicht, indem deren prototypische Implementierung einem umfassenden Performancetest unterworfen wurde. Dafür erfolgte zu Beginn die Festlegung allgemeiner Testbedingungen hin-

sichtlich der verwendeten Hardware, Datenbanksysteme und Testdaten. Weiterhin wurden die relevanten Anfragen klassifiziert und mit Bezug zum Anwendungsbeispiel definiert. Deren Umsetzung fand schließlich nicht nur für die API statt, sondern zusätzlich zur besseren Vergleichbarkeit für zwei weitere SQL-Testkandidaten. Mit Hilfe eines Tools wurde die eigentliche Durchführung des Tests automatisiert. Die hierbei ermittelten Antwortzeiten waren die Grundlage zur qualitativen sowie quantitativen Bewertung der Zugriffsschicht. Abgesehen von wenigen Ausnahmen zeigte sich dabei die API gegenüber den beiden SQL-Testkandidaten nicht nur um Größenordnungen performanter, sondern lieferte meist auch Ergebnisse innerhalb typischer Grenzwerte für interaktives Arbeiten.

7.2 Weiterführende Arbeiten

Dieser Abschnitt gibt einen Ausblick auf relevante angrenzende Themen, die in zukünftigen Arbeiten genauer analysiert und bearbeitet werden können.

Erweiterung des Referenzmodells

Die relationalen Strukturen des in Abschnitt 4.3 vorgestellten Referenzmodells dienen der Verwaltung aspektspezifischer Daten. Allerdings können gemäß Abschnitt 4.1 derartige Informationen nur für Attribute zu bereits *existierenden* Datensätzen des fachlichen Datenmodells hinterlegt werden. Prinzipiell ist entsprechend der Betrachtungen in Abschnitt 3.1.2 aber auch die Existenz vollständiger Datensätze unter funktionalen Aspekten vorstellbar. Insbesondere für die Nutzung des AOD-Paradigmas in Anwendungsdomänen neben Produktkonfiguratoren ist eine diesbezügliche Erweiterung des Modells sowie der Referenzimplementierung vorstellbar. Grundlage eines möglichen Lösungsansatzes ist die Persistierung aspektspezifischer Datensätze direkt in den fachlichen Tabellen. Zusätzlich muss in einer neuen Verwaltungstabelle AspectAssignRows definiert werden, ob ein mittels RowID referenzierter Datensatz für eine Aspektausprägung gültig ist oder nicht. Hierbei ließe sich das Konzept der dreiwertigen Logik anwenden, wie es in [Göb09] (siehe dort Abschnitt 3.6.3) beschrieben ist.

Erweiterung der Zugriffsschicht

Durch die bewusste Fokussierung auf eine *prototypische* Implementierung der Zugriffsschicht existiert für diese zwangsläufig Erweiterungspotential. Hinsichtlich Benutzbarkeit und Flexibilität ist ein zentrales Thema unter anderem die derzeitige Beschränkung auf *eine* fachliche Relation je Anfrage gemäß der in Abschnitt 5.2.1 definierten Methodensignaturen von *createModifyStatement* und *createQueryStatement*. Zur Unterstützung beliebig vieler Tabellen müssten Verknüpfungs-Operatoren nach dem Vorbild des SQL-Verbunds spezifiziert und entsprechend im Zugriffsmodell verarbeitet werden. Darüber hinaus ließen sich sowohl die Wertfilter-Sprache als auch die Syntax für Aspektfilter bezüglich ihrer Mächtigkeit erweitern. Einerseits könnten dazu beispielsweise für die in Abschnitt 5.1.2.2 erläuterten Wertfilter-Ausdrücke zusätzliche mathematische Operatoren eingeführt werden. Andererseits stellt die Aufhebung der Einschränkung auf nur ein aspektspezifisches Attribut je Vergleichs-Prädikat sowohl für Wertfilter als auch Aspektfilter eine mögliche strategische Weiterentwicklung dar.

Optimierung der Zugriffsschicht

Wie die Auswertung der Testergebnisse in Abschnitt 6.5 gezeigt hat, lieferte die Zugriffsschicht im Vergleich zu den anderen beiden Testkandidaten die Ergebnisse in den meisten Fällen mit einer deutlich kürzeren Antwortzeit. Allerdings sind diese insbesondere für modifizierende Anfragen und auf größeren Datenbeständen auch bereits im zweistelligen Sekundenbereich. Für eine weitere Beschleunigung bieten sich zwei voneinander unabhängige Optimierungs-Strategien an. Einerseits lässt sich die Komponente zur Vorverarbeitung von Filterbedingungen mit zusätzlicher Logik anreichern. Dadurch könnten beispielsweise unnötige und aufwändige Transformationen der Daten wie im Fall von `MD+*09` in Abschnitt 6.5.2 entfallen. Allerdings gibt es selbst bei einer optimalen Vorverarbeitung für die Datenbanklast und das zu transportierende Datenvolumen eine durch das Persistenzmodell definierte untere Schranke. Andererseits können zusätzliche Beschleunigungseffekte mit Hilfe intelligenter Caching-Strategien gemäß Abschnitt 5.1.1 erzielt werden. Abhängig von der Struktur des Zwischenspeichers ließe sich nicht nur der Datentransfer vom DBMS zur Zugriffsschicht reduzieren, sondern auch die Transformation der für eine Anfrage relevanten Daten. Die hiermit erreichbaren Verbesserungen der Antwortzeiten um einige Größenordnungen erfordern jedoch einen relativ hohen Implementierungsaufwand zur Sicherstellung von Konsistenzanforderungen.

Analyse mehrdimensionaler Datenstrukturen

Aufgrund der in Abschnitt 3.1.2 beschriebenen Charakteristik funktionaler Aspekte erfordert die zuvor erwähnte Optimierung durch Caching-Strategien den Einsatz mehrdimensionaler Datenstrukturen. Vor deren Verwendung in der Zugriffsschicht-Implementierung sollte eine fundierte Analyse und Bewertung relevanter theoretischer Konzepte wie des (mehrstufigen) Gridfile [NHS84] oder des PLOP Hashing [KS88] erfolgen. Darüber hinaus könnten die zusätzlichen Berechnungs-Ressourcen in der Graphics Processing Unit (GPU) genutzt werden, um beispielsweise notwendige Hashtabellen-Lookups [ASA+09] durchzuführen oder mehrdimensionale Zugriffsstrukturen zu verwalten [ZHWG08].

Anhang A

Definition „Produktkonfigurator“

Nachfolgend sind verschiedene Ansätze zur Definition des Begriffs eines Produktkonfigurators mit Verweis auf die jeweilige Fachliteratur aufgeführt.

„Die Produktkonfiguration beschreibt das Zusammensetzen eines Produktes aus vorgegebenen Produktkomponenten (sogenannte Selektion und Kombination) und die Selektion inhaltlicher Ausprägungen der Komponenteneigenschaften (sogenannte Parametrisierung) unter Einhaltung der Konfigurationsregeln. Die Konfigurationsmöglichkeiten ergeben sich aus den Selektions-, Kombinations- und Parametrisierungsmöglichkeiten eines Produktes eingeschränkt durch die Konfigurationsregeln.“ [Sch06]

„Produktkonfigurator: Ein Werkzeug, das hilft ein Produkt so zu bestimmen, dass es vorgegebenen Eigenschaften genügt. Ein Produktkonfigurator kann auf verschiedene Weise erstellt werden, er kann speziell programmiert werden oder es kann ein Werkzeug zu seiner Erstellung benutzt werden. Die Software zur Erstellung eines Produktkonfigurators wird als Konfigurationssoftware bezeichnet.“ [Bri99]

„Produktkonfiguratoren sind multifunktionale, rechnergestützte Systeme, die als Schnittstelle zwischen Vertrieb und wertschöpfungsnahen Funktionen stehen. Sie dienen zur informationstechnischen Wissens- und Aufgabenintegration mit dem Ziel, die Verkaufs- und Aufgabenabwicklungsprozesse effektiv und effizient zu unterstützen.“ [KRA02]

„Ein Produktkonfigurator ist ein computergestütztes System, mit dem der Kunde typischerweise über das Internet bzw. World Wide Web interagiert. [. . .] Der Konfigurator bietet dem Kunden die Möglichkeit, im Rahmen eines Konfigurationsvorgangs die gewünschten Produktmodule zu wählen sowie die Produktparameter nach Wunsch zu konkretisieren und wacht dabei über die Einhaltung der Produktregeln.“ [Pol08]

„Konfigurationssysteme stellen [. . .] ein integrales Bindeglied zwischen Produktentwicklung, Fertigung und Kundenwunsch dar. Ausgestattet mit einer einfachen Benutzerschnittstelle leiten diese Systeme den Kunden (und gegebenenfalls einen Mitarbeiter im Verkauf) durch die Erhebung der Bedürfnisinformation – und prüfen sogleich die Konsistenz sowie die Fertigungsfähigkeit der gewünschten Variante.“ [RP06]

Anhang B

ANTLR3-Grammatiken

```
grammar IAF;

options {
  output=AST;
}

tokens {
  ASPECT = 'ASPECT';
  AND = 'AND';
  OR = 'OR';
  NOT = 'NOT';
  IS = 'IS';
  IN = 'IN';
  NULL = 'NULL';
  EQ = '=';
  LPAREN = '(';
  RPAREN = ')';
  COMMA = ',';
}

@lexer::header{
  package de.unijena.dbis.aodm.refimpl;
}

@header {
  package de.unijena.dbis.aodm.refimpl;
  import java.util.HashMap;
  import java.util.Arrays;
  import de.unijena.dbis.aodm.AspectFilter;
  import de.unijena.dbis.aodm.AspectManager;
}

@members {
  AspectManager am;

  public IAFParser(AspectManager am, TokenStream input) {
    this(input, new RecognizerSharedState());
    this.am = am;
  }
```

```
public IAFParser(AspectManager am, TokenStream input, RecognizerSharedState state) {
  this(input, state);
  this.am = am;
}

public static IAFParser createIAFParser(AspectManager am, String infixAF) {
  IAFLexer lex = new IAFLexer(new ANTLRStringStream(infixAF));
  CommonTokenStream tokens = new CommonTokenStream(lex);
  IAFParser p = new IAFParser(am, tokens);
  return p;
}

public AspectFilter.FilterNode buildAF() throws org.antlr.runtime.RecognitionException,
      de.unijena.dbis.aodm.AspectLookupException {
  start_return r = start();
  return buildAF((Tree) r.tree);
}

public int lookupKey(Tree t) throws de.unijena.dbis.aodm.AspectLookupException {
  switch (t.getType()) {
  case IAFParser.NUMBER:
    return Integer.parseInt(t.getText());
  case IAFParser.QUOTED_CHARS:
    String s = t.getText();
    return am.getAspectCatalogManager().lookupAspectID(s.substring(1, s.length()-1));
  }
  return 0;
}

public int lookupValue(int key, Tree t) throws de.unijena.dbis.aodm.AspectLookupException {
  switch (t.getType()) {
  case IAFParser.NUMBER:
    return Integer.parseInt(t.getText());
  case IAFParser.QUOTED_CHARS:
    String s = t.getText();
    return am.getAspectCatalogManager().lookupAspectKeyValueID(key,
          s.substring(1, s.length()-1));
  }
  return 0;
}

AspectFilter.FilterNode buildAF(Tree t) throws de.unijena.dbis.aodm.AspectLookupException {
  AspectFilter.FilterNode fn = null;
  if (t == null) return null;
  switch (t.getType()) {
  case IAFParser.OR:
    List <AspectFilter.FilterNode> ops =
        new ArrayList <AspectFilter.FilterNode> (t.getChildCount());
    int j = 0;
    for (int i = 0; i < t.getChildCount(); i++) {
      AspectFilter.FilterNode n = buildAF (t.getChild(i));
      if (n instanceof AspectFilterImpl.Junction &&
          ((AspectFilterImpl.Junction) n).type == AspectFilterImpl.Junction.Type.OR) {
        AspectFilterImpl.Junction junc = (AspectFilterImpl.Junction) n;
        for (int k = 0; k < junc.ops.length; k++) {
          ops.add(junc.ops[k]);
        }
```

```
        }
        else ops.add(n);
      }
      return AspectFilterImpl.or(ops.toArray(new AspectFilter.FilterNode[] {}));

    case IAFParser.AND:
      List <AspectFilter.FilterNode> ops =
            new ArrayList <AspectFilter.FilterNode> (t.getChildCount());
      int j = 0;
      for (int i = 0; i < t.getChildCount(); i++) {
        AspectFilter.FilterNode n = buildAF (t.getChild(i));
        if (n instanceof AspectFilterImpl.Junction &&
            ((AspectFilterImpl.Junction) n).type == AspectFilterImpl.Junction.Type.AND) {
          AspectFilterImpl.Junction junc = (AspectFilterImpl.Junction) n;
          for (int k = 0; k < junc.ops.length; k++) {
            ops.add(junc.ops[k]);
          }
        }
        else ops.add(n);
      }
      return AspectFilterImpl.and(ops.toArray(new AspectFilter.FilterNode[] {}));

    case IAFParser.NOT:
      AspectFilter.FilterNode[] ops = new AspectFilter.FilterNode[t.getChildCount()];
      for (int i = 0; i < t.getChildCount(); i++) {
        ops[i] = buildAF(t.getChild(i));
      }
      return AspectFilterImpl.not(ops[0]);

    case IAFParser.EQ:
      int key = lookupKey(t.getChild(0));
      return AspectFilterImpl.keyValue(key, lookupValue(key, t.getChild(1)));

    case IAFParser.IN:
      if (t.getChildCount() > 2) {
        int key = lookupKey (t.getChild(0));
        AspectFilter.FilterNode[] ops = new AspectFilter.FilterNode[t.getChildCount()-1];
        for (int i = 1; i < t.getChildCount(); i++) {
          int value = lookupValue(key, t.getChild(i));
          ops[i-1] = AspectFilterImpl.keyValue(key, value);
        }
        return AspectFilterImpl.or(ops);
      }
      else {
        int key = lookupKey(t.getChild(0));
        return AspectFilterImpl.keyValue(key, lookupValue(key, t.getChild(1)));
      }

    case IAFParser.IS:
      int key = lookupKey(t.getChild(0));
      if (t.getChild(1).getType() == IAFParser.NULL)
        return AspectFilterImpl.keyNull(key);
      else if (t.getChild(1).getType() == IAFParser.NOT &&
               t.getChild(2).getType() == IAFParser.NULL)
        return AspectFilterImpl.not(AspectFilterImpl.keyNull(key));
    }
```

```
    return null;
  }
}

// ---------- PARSER RULES ----------
start : EOF! | (expr EOF!);
expr : conj (OR^ conj)*;
conj : atom (AND^ atom)*;
atom : assign
    | NOT^ LPAREN! expr RPAREN!
    | NOT^ assign
    | LPAREN! expr RPAREN!;
assign : key ((EQ^ value)
         | (IS^ (NOT)? NULL)
         | (IN^ LPAREN! value (COMMA! value)* RPAREN!));
key : (QUOTED_CHARS) | (ASPECT! NUMBER);
value : (QUOTED_CHARS) | (NUMBER);

// ---------- LEXER RULES ----------
NUMBER : (DIGIT)+ ;
QUOTED_CHARS : '\'' ( '\'\'' | ~('\'') )* '\'';
WHITESPACE : ( '\t' | ' ' | '\r' | '\n'| '\u000C' )+  { $channel = HIDDEN; };
fragment DIGIT : '0'..'9';
```

Listing B.1: ANTLR3-Grammatik für Infix-Aspektfilter

```
grammar VF;

options {
  output = AST;
}

tokens {
  AND = 'AND';
  OR = 'OR';
  NOT = 'NOT';
  IS = 'IS';
  IN = 'IN';
  NULL = 'NULL';
  EQ = '=';
  NE = '<>';
  LT = '<';
  GT = '>';
  LE = '<=';
  GE = '>=';
  LIKE = 'LIKE';
  LPAREN = '(';
  RPAREN = ')';
  COMMA = ',';
}

@lexer::header{
  package de.unijena.dbis.aodm.refimpl;
}

@header {
  package de.unijena.dbis.aodm.refimpl;
```

```
  import de.unijena.dbis.aodm.AspectManager;
  import de.unijena.dbis.aodm.refimpl.ValueFilterImpl;
}

@members {
  AspectManager am;
  String defaultTable = null;
  int tabID = -1;

  public VFParser(AspectManager am, TokenStream input) {
    this(input, new RecognizerSharedState());
    this.am = am;
  }

  public VFParser(AspectManager am, TokenStream input, RecognizerSharedState state) {
    this(input, state);
    this.am = am;
  }

  public void setDefaultTable(String tab, int tabID) {
    this.defaultTable = tab;
    this.tabID = tabID;
  }

  public static VFParser createParser(AspectManager am, String valueFilterString) {
    VFLexer lex = new VFLexer(new ANTLRStringStream(valueFilterString));
    CommonTokenStream tokens = new CommonTokenStream(lex);
    VFParser p = new VFParser(am, tokens);
    return p;
  }

  public ValueFilterImpl.FilterNode buildFilter() throws org.antlr.runtime.RecognitionException,
          de.unijena.dbis.aodm.AspectLookupException {
    start_return r = start();
    return buildFilter((Tree) r.tree);
  }

  ValueFilterImpl.Value buildFunc(Tree t, ValueFilterImpl.Value[] vals) throws
        de.unijena.dbis.aodm.AspectLookupException {
    switch (t.getType()) {
    case IDENTIFIER:
      return new ValueFilterImpl.Function(t.getText(), vals);
    }
    return null;
  }

  ValueFilterImpl.Value buildValue(Tree t) throws de.unijena.dbis.aodm.AspectLookupException {
    switch (t.getType()) {
    case LPAREN:
      ValueFilterImpl.Value[] vals = new ValueFilterImpl.Value[t.getChildCount()-1];
      for (int i = 0; i < t.getChildCount()-1; i++) {
        vals[i] = buildValue(t.getChild(i+1));
      }
      return buildFunc(t.getChild(0), vals);

    case IDENTIFIER:
      int colID = am.getAspectCatalogManager().lookupColumnID(tabID, t.getText());
```

```
      return new ValueFilterImpl.ColumnValue(defaultTable, t.getText(), tabID, colID);

    case NUMBER:
      return new ValueFilterImpl.NumberValue(t.getText());

    case QUOTED_CHARS:
      return new ValueFilterImpl.StringValue(t.getText());
    }

    return null;
  }

  ValueFilterImpl.FilterNode buildFilter(Tree t) throws
        de.unijena.dbis.aodm.AspectLookupException {
    ValueFilterImpl.FilterNode fn = null;
    if (t == null) return null;
    switch (t.getType()) {
    case VFParser.OR:
      List <ValueFilterImpl.FilterNode> ops = new ArrayList
            <ValueFilterImpl.FilterNode> (t.getChildCount());
      int j = 0;
      for (int i = 0; i < t.getChildCount(); i++) {
        ValueFilterImpl.FilterNode n = buildFilter (t.getChild(i));
        if (n instanceof ValueFilterImpl.Junction &&
            ((ValueFilterImpl.Junction) n).type == ValueFilterImpl.Junction.Type.OR) {
          ValueFilterImpl.Junction junc = (ValueFilterImpl.Junction) n;
          for (int k = 0; k < junc.ops.length; k++) {
            ops.add(junc.ops[k]);
          }
        }
        else
          ops.add(n);
      }
      return new ValueFilterImpl.Junction (ValueFilterImpl.Junction.Type.OR,
                                           ops.toArray(new ValueFilterImpl.FilterNode[] {}));

    case VFParser.AND:
      List <ValueFilterImpl.FilterNode> ops = new ArrayList
            <ValueFilterImpl.FilterNode> (t.getChildCount());
      int j = 0;
      for (int i = 0; i < t.getChildCount(); i++) {
        ValueFilterImpl.FilterNode n = buildFilter (t.getChild(i));
        if (n instanceof ValueFilterImpl.Junction &&
            ((ValueFilterImpl.Junction) n).type == ValueFilterImpl.Junction.Type.AND) {
          ValueFilterImpl.Junction junc = (ValueFilterImpl.Junction) n;
          for (int k = 0; k < junc.ops.length; k++) {
            ops.add(junc.ops[k]);
          }
        }
        else
          ops.add(n);
      }
      return new ValueFilterImpl.Junction(ValueFilterImpl.Junction.Type.AND,
                                          ops.toArray(new ValueFilterImpl.FilterNode[] {}));

    case VFParser.NOT:
      ValueFilterImpl.FilterNode[] ops = new ValueFilterImpl.FilterNode[t.getChildCount()];
```

```
    for (int i = 0; i < t.getChildCount(); i++) {
      ops[i] = buildFilter(t.getChild(i));
    }
    return new ValueFilterImpl.Junction(ValueFilterImpl.Junction.Type.NOT, ops[0]);

  case VFParser.EQ:
    return new ValueFilterImpl.ComparisionPredicate(
      ValueFilterImpl.ComparisionPredicate.OpType.EQUAL,
      buildValue(t.getChild(0)), buildValue(t.getChild(1)));

  case VFParser.NE:
    return new ValueFilterImpl.ComparisionPredicate(
      ValueFilterImpl.ComparisionPredicate.OpType.UNEQUAL,
      buildValue (t.getChild(0)), buildValue(t.getChild(1)));

  case VFParser.LT:
    return new ValueFilterImpl.ComparisionPredicate(
      ValueFilterImpl.ComparisionPredicate.OpType.LOWER,
      buildValue(t.getChild(0)), buildValue(t.getChild(1)));

  case VFParser.GT:
    return new ValueFilterImpl.ComparisionPredicate(
      ValueFilterImpl.ComparisionPredicate.OpType.GREATER,
      buildValue(t.getChild(0)), buildValue(t.getChild(1)));

  case VFParser.LE:
    return new ValueFilterImpl.ComparisionPredicate(
      ValueFilterImpl.ComparisionPredicate.OpType.LOWER_EQUAL,
      buildValue(t.getChild(0)), buildValue(t.getChild(1)));

  case VFParser.GE:
    return new ValueFilterImpl.ComparisionPredicate(
      ValueFilterImpl.ComparisionPredicate.OpType.GREATER_EQUAL,
      buildValue(t.getChild(0)), buildValue(t.getChild(1)));

  case VFParser.LIKE:
    return new ValueFilterImpl.ComparisionPredicate(
      ValueFilterImpl.ComparisionPredicate.OpType.LIKE,
      buildValue(t.getChild(0)), buildValue(t.getChild(1)));

  case VFParser.IS:
    ValueFilterImpl.Value v = buildValue(t.getChild(0));
    if (t.getChild(1).getType() == VFParser.NULL) {
      return new ValueFilterImpl.ComparisionPredicate(
        ValueFilterImpl.ComparisionPredicate.OpType.IS_NULL, v);
    }
    else if (t.getChild(1).getType() == VFParser.NOT &&
             t.getChild(2).getType() == VFParser.NULL) {
      return new ValueFilterImpl.ComparisionPredicate(
        ValueFilterImpl.ComparisionPredicate.OpType.IS_NOT_NULL, v);
    }

  case VFParser.IN:
    List <ValueFilterImpl.FilterNode> ops = new ArrayList
         <ValueFilterImpl.FilterNode> (t.getChildCount());
    int j = 0;
    ValueFilterImpl.Value v0 = buildValue(t.getChild(0));
```

```
        for (int i = 1; i < t.getChildCount(); i++) {
          ValueFilterImpl.FilterNode n = ValueFilterImpl.ComparisionPredicate(
            ValueFilterImpl.ComparisionPredicate.OpType.EQUAL, v0, buildValue(t.getChild(i)));
          ops.add(n);
        }
        return new ValueFilterImpl.Junction(ValueFilterImpl.Junction.Type.OR,
          ops.toArray (new ValueFilterImpl.FilterNode[] {}));
      }

      return null;
    }
}

// ---------- PARSER RULES ----------
@rulecatch {
  catch (RecognitionException rex) {
    throw rex;
  }
}

start : EOF! | (expr EOF!);
expr : conj (OR^ conj)*;
conj : atom (AND^ atom)*;
atom : comp_pred
     | NOT^ LPAREN! expr RPAREN!
     | NOT^ comp_pred
     | LPAREN! expr RPAREN!;
comp_pred :
     value (((EQ | NE | LT | GT | LE | GE | LIKE)^ rvalue)
          | (IS^ (NOT)? NULL)
          | (IN^ LPAREN! rvalue (COMMA! rvalue)* RPAREN!));
value :
     IDENTIFIER
     | NUMBER
     | QUOTED_CHARS
     | func LPAREN^ value (COMMA! value)* RPAREN!;
rvalue :
    NUMBER | QUOTED_CHARS
     | func LPAREN^ rvalue (COMMA! rvalue)* RPAREN!;
func :
     IDENTIFIER;

// ---------- LEXER RULES ----------
IDENTIFIER : LETTER (LETTER | DIGIT | '_')+;
NUMBER : (DIGIT)+ ('.' (DIGIT)+)? (('e'|'E') (DIGIT)+)?;
QUOTED_CHARS : '\'' ( '\'\'' | ~('\'') )* '\'';
WHITESPACE : ( '\t' | ' ' | '\r' | '\n'| '\u000C' )+  { $channel = HIDDEN; } ;
fragment DIGIT : '0'..'9' ;
fragment LETTER : ('a'..'z'|'A'..'Z');
```

Listing B.2: ANTLR3-Grammatik für Wertfilter

Anhang C

SQL-Skripte im Testumfeld

Die nachfolgenden Listings stellen die notwendigen SQL-Skripte für den in Kapitel 6 beschriebenen Performancetest unter DB2 dar. Analog wurden auch entsprechende Skripte für die Verarbeitung durch PostgreSQL erstellt, die in [Pie11b] zu finden sind.

C.1 Basistabelle und Aspektkatalog im Testszenario

```
CREATE TABLE Module (
    Key VARCHAR(100) NOT NULL,
    Name VARCHAR(70) NOT NULL,
    Price NUMERIC(8,2) NOT NULL,
    Description VARCHAR(70) NOT NULL,
    Norm VARCHAR(70),
    Material VARCHAR(70),
    UserCreate VARCHAR(20),
    DateCreate DATE,
    RowID INTEGER NOT NULL GENERATED BY DEFAULT AS IDENTITY UNIQUE,
    PRIMARY KEY (ID)
);
```

Listing C.1: SQL-Skript für die Basistabelle MODULE

```
CREATE TABLE AspectTable (
    AspTabID INTEGER NOT NULL GENERATED BY DEFAULT AS IDENTITY,
    Schema VARCHAR(100) NOT NULL,
    TableName VARCHAR(100) NOT NULL,
    IDColName VARCHAR(100) NOT NULL DEFAULT 'RowID',
    PRIMARY KEY (AspTabID)
);

CREATE TABLE AspectDatatype (
    AspTypeID INTEGER NOT NULL GENERATED BY DEFAULT AS IDENTITY,
    TypeName VARCHAR(100) NOT NULL,
    Length INT NOT NULL,
    Scale INT,
    PRIMARY KEY (AspTypeID)
);
```

```
CREATE TABLE AspectColumn (
    AspColID INTEGER NOT NULL GENERATED BY DEFAULT AS IDENTITY,
    Tab INTEGER REFERENCES AspectTable (AspTabID) NOT NULL,
    ColumnName VARCHAR(100) NOT NULL,
    Datatype INTEGER REFERENCES AspectDatatype (AspTypeID) NOT NULL,
    UNIQUE (AspColID, Tab),
    PRIMARY KEY (AspColID)
);

CREATE TABLE AspectDefinition (
    AspDefID INTEGER NOT NULL GENERATED BY DEFAULT AS IDENTITY,
    Name VARCHAR(100) NOT NULL,
    Key VARCHAR(100) NOT NULL,
    Datatype INTEGER REFERENCES AspectDatatype (AspTypeID) NOT NULL,
    PRIMARY KEY (AspDefID)
);

CREATE TABLE AspectKeyValue (
    AspKeyID INTEGER NOT NULL GENERATED BY DEFAULT AS IDENTITY,
    Aspect INTEGER REFERENCES AspectDefinition (AspDefID) NOT NULL,
    KeyValue VARCHAR(200) NOT NULL,
    Comment VARCHAR(200),
    UNIQUE (AspKeyID, Aspect),
    PRIMARY KEY (AspKeyID)
);

CREATE TABLE AspectValue (
    AspValID INTEGER NOT NULL GENERATED BY DEFAULT AS IDENTITY,
    Tab INTEGER NOT NULL,
    Col INTEGER NOT NULL,
    RowID INTEGER NOT NULL,
    Value VARCHAR(255),
    FOREIGN KEY (Col, Tab) REFERENCES AspectColumn (AspColID, Tab),
    PRIMARY KEY (AspValID)
);

CREATE TABLE AspectAssign (
    AspAssID INTEGER NOT NULL GENERATED BY DEFAULT AS IDENTITY,
    KeyValue INTEGER NOT NULL,
    Aspect INTEGER NOT NULL,
    AspectValue INTEGER NOT NULL REFERENCES AspectValue (AspValID)
                                  ON DELETE CASCADE,
    FOREIGN KEY (KeyValue, Aspect) REFERENCES AspectKeyValue (AspKeyID, Aspect),
    PRIMARY KEY (AspAssID)
);

CREATE TABLE AspectDependence (
    ColID INTEGER NOT NULL REFERENCES AspectColumn (AspColID),
    AspID INTEGER NOT NULL REFERENCES AspectDefinition (AspDefID),
    PRIMARY KEY (ColID, AspID)
);
CREATE INDEX AspectValueRowIDIndex ON AspectValue (RowID);
CREATE INDEX AspectAssignValueIDIndex ON AspectAssign (AspectValue);
CREATE INDEX AspectValueTabIndex ON AspectValue (Tab);
```

Listing C.2: SQL-Skript für die Aspektkatalog-Struktur

```
INSERT INTO AspectDatatype (AspTypeID, Typename, Length, Scale)
VALUES (501, 'INT', 8, NULL),
       (502, 'STRING', 5, NULL),
       (503, 'STRING', 10, NULL),
       (504, 'STRING', 100, NULL),
       (505, 'STRING', 70, NULL),
       (506, 'NUMERIC', 8, 2);

INSERT INTO AspectDefinition (AspDefID, Name, Key, Datatype)
VALUES (101, 'I18N/Language', 'Language', 502),
       (102, 'I18N/Region', 'Region', 502),
       (103, 'Versioning', 'Version', 501),
       (104, 'Pricing', 'Pricegrade', 502);

INSERT INTO AspectKeyValue (AspKeyID, Aspect, KeyValue, Comment)
VALUES (201, 101, 'en', NULL), (202, 101, 'en_US', NULL), (203, 101, 'en_UK', NULL),
       (204, 101, 'de', NULL), (205, 101, 'de_DE', NULL), (206, 101, 'de_CH', NULL),
       (207, 101, 'fr', NULL), (208, 101, 'fr_FR', NULL), (209, 101, 'fr_CH', NULL),
       (210, 101, 'it', NULL), (211, 101, 'it_CH', NULL), (217, 101, 'it_IT', NULL),
       (212, 101, 'po', NULL), (213, 101, 'zh', NULL), (214, 101, 'zh_CN', NULL),
       (215, 101, 'zh_HK', NULL), (216, 101, 'zh_TW', NULL);

INSERT INTO AspectKeyValue (AspKeyID, Aspect, KeyValue, Comment)
VALUES (301, 102, 'UK', NULL), (302, 102, 'EURO', NULL), (303, 102, 'USA', NULL),
       (304, 102, 'GER', NULL), (305, 102, 'SUI', NULL), (306, 102, 'AUT', NULL),
       (307, 102, 'FRA', NULL), (308, 102, 'ITA', NULL), (309, 102, 'POL', NULL),
       (310, 102, 'CHN', NULL), (311, 102, 'TPE', NULL);

INSERT INTO AspectKeyValue (AspKeyID, Aspect, KeyValue, Comment)
VALUES (401, 103, '1', NULL), (402, 103, '2', NULL), (403, 103, '3', NULL),
       (404, 103, '4', NULL), (405, 103, '5', NULL), (406, 103, '6', NULL),
       (407, 103, '7', NULL), (408, 103, '8', NULL), (409, 103, '9', NULL),
       (410, 103, '10', NULL), (411, 103, '11', NULL), (412, 103, '12', NULL),
       (413, 103, '13', NULL), (414, 103, '14', NULL), (415, 103, '15', NULL);

INSERT INTO AspectKeyValue (AspKeyID, Aspect, KeyValue, Comment)
VALUES (501, 104, 'standard', NULL), (502, 104, 'PG_1', NULL), (503, 104, 'PG_2', NULL),
       (504, 104, 'PG_3', NULL), (505, 104, 'PG_4', NULL), (506, 104, 'PG_5', NULL),
       (507, 104, 'PG_6', NULL), (508, 104, 'PG_7', NULL), (509, 104, 'PG_8', NULL);

INSERT INTO AspectTable (AspTabID, Schema, TableName, IDColName)
VALUES (1001, $caSchema, 'Module', 'RowID'), (1009, $caSchema, 'Aux', 'ID');

INSERT INTO AspectColumn (AspColID, Tab, ColumnName, Datatype)
VALUES (1101,1001, 'Name', 505), (1102,1001, 'Price', 506), (1103,1001, 'Description', 505),
       (1104,1001, 'Norm', 505), (1105,1001, 'Material', 505), (1191,1009, 'Junk', 504);

INSERT INTO AspectDependence (ColID, AspID)
VALUES (1101, 101), (1101, 103),
       (1102, 102), (1102, 103), (1102, 104),
       (1103, 101), (1103, 103),
       (1104, 102), (1104, 103),
       (1105, 101), (1105, 103),
       (1191, 101), (1191, 102), (1191, 103), (1191, 104);
```

Listing C.3: SQL-Skript für die Aspektkatalog-Metadaten

C.2 Workload-Realisierung mit gSQL

```
CREATE VIEW ContextualValues AS
SELECT
    RowID, Col, AspValID AS ValID,
    COALESCE (av1.Tab, av2.Tab, av3.Tab, av4.Tab) AS Tab,
    COALESCE (av1.Value, av2.Value, av3.Value, av4.Value) AS Value,
    aa1.KeyValue AS kv1,
    aa2.KeyValue AS kv2,
    aa3.KeyValue AS kv3,
    aa4.KeyValue AS kv4
FROM (AspectValue av1 JOIN AspectAssign aa1 ON
      aa1.Aspect = 101 AND aa1.AspectValue = av1.AspValID)
FULL OUTER JOIN (AspectValue av2 JOIN AspectAssign aa2 ON
                 aa2.Aspect = 102 AND aa2.AspectValue = av2.AspValID)
     ON av1.RowID = av2.RowID
    AND av1.Col = av2.Col
    AND av1.AspValID = av2.AspValID
FULL OUTER JOIN (AspectValue av3 JOIN AspectAssign aa3 ON
                 aa3.Aspect = 103 AND aa3.AspectValue = av3.AspValID)
     ON COALESCE (av1.RowID, av2.RowID) = av3.RowID
    AND COALESCE (av1.Col, av2.Col) = av3.Col
    AND COALESCE (av1.AspValID, av2.AspValID) = av3.AspValID
FULL OUTER JOIN (AspectValue av4 JOIN AspectAssign aa4 ON
                 aa4.Aspect = 104 AND aa4.AspectValue = av4.AspValID)
     ON COALESCE (av1.RowID, av2.RowID, av3.RowID) = av4.RowID
    AND COALESCE (av1.Col, av2.Col, av3.Col) = av4.Col
    AND COALESCE (av1.AspValID, av2.AspValID, av3.AspValID) = av4.AspValID
```

Listing C.4: SQL-View ContextualValues für signierte Tupel (ohne USING)

```
CREATE VIEW V_Module AS
SELECT
    RowID,
    kv1 AS Language, kv2 AS Region, kv3 AS Version, kv4 AS Pricegrade,
    col1.Value AS Name,
    col2.Value AS Price,
    col3.Value AS Description,
    col4.Value AS Norm,
    col5.Value AS Material
FROM (SELECT * FROM ContextualValues WHERE Tab=1001 AND Col=1101)
  AS col1
FULL OUTER JOIN (SELECT * FROM ContextualValues WHERE Tab=1001 AND Col=1102)
  AS col2
  ON col1.RowID=col2.RowID
 AND COALESCE (col1.kv1,-1)=COALESCE (col2.kv1,-1)
 AND COALESCE (col1.kv2,-1)=COALESCE (col2.kv2,-1)
 AND COALESCE (col1.kv3,-1)=COALESCE (col2.kv3,-1)
 AND COALESCE (col1.kv4,-1)=COALESCE (col2.kv4,-1)
FULL OUTER JOIN (SELECT * FROM ContextualValues WHERE Tab=1001 AND Col=1103)
  AS col3
  ON COALESCE (col1.RowID,col2.RowID)=col3.RowID
 AND COALESCE (col1.kv1,col2.kv1,-1)=COALESCE (col3.kv1,-1)
 AND COALESCE (col1.kv2,col2.kv2,-1)=COALESCE (col3.kv2,-1)
 AND COALESCE (col1.kv3,col2.kv3,-1)=COALESCE (col3.kv3,-1)
 AND COALESCE (col1.kv4,col2.kv4,-1)=COALESCE (col3.kv4,-1)
```

```
FULL OUTER JOIN (SELECT * FROM ContextualValues WHERE Tab=1001 AND Col=1104)
  AS col4
  ON COALESCE (col1.RowID,col2.RowID,col3.RowID)=col4.RowID
 AND COALESCE (col1.kv1,col2.kv1,col3.kv1,-1)=COALESCE (col4.kv1,-1)
 AND COALESCE (col1.kv2,col2.kv2,col3.kv2,-1)=COALESCE (col4.kv2,-1)
 AND COALESCE (col1.kv3,col2.kv3,col3.kv3,-1)=COALESCE (col4.kv3,-1)
 AND COALESCE (col1.kv4,col2.kv4,col3.kv4,-1)=COALESCE (col4.kv4,-1)
FULL OUTER JOIN (SELECT * FROM ContextualValues WHERE Tab=1001 AND Col=1105)
  AS col5
  ON COALESCE (col1.RowID,col2.RowID,col3.RowID,col4.RowID)=col5.RowID
 AND COALESCE (col1.kv1,col2.kv1,col3.kv1,col4.kv1,-1)=COALESCE (col5.kv1,-1)
 AND COALESCE (col1.kv2,col2.kv2,col3.kv2,col4.kv2,-1)=COALESCE (col5.kv2,-1)
 AND COALESCE (col1.kv3,col2.kv3,col3.kv3,col4.kv3,-1)=COALESCE (col5.kv3,-1)
 AND COALESCE (col1.kv4,col2.kv4,col3.kv4,col4.kv4,-1)=COALESCE (col5.kv4,-1)
```

Listing C.5: SQL-View V_MODULE für aspektspezifische Produktdaten (ohne USING)

```
CREATE VIEW V_Module_Derived AS
SELECT
    RowID,
    kv1 AS Language, kv2 AS Region, kv3 AS Version, kv4 AS Pricegrade,
    col1.Value AS Name,
    col2.Value AS Price,
    col3.Value AS Description,
    col4.Value AS Norm,
    col5.Value AS Material
FROM (SELECT * FROM ContextualValues WHERE Tab=1001 AND Col=1101)
  AS col1
FULL OUTER JOIN (SELECT * FROM ContextualValues WHERE Tab=1001 AND Col=1102)
  AS col2
  ON col1.RowID=col2.RowID
 AND COALESCE (col1.kv1,col2.kv1,-1)=COALESCE (col2.kv1,col1.kv1,-1)
 AND COALESCE (col1.kv2,col2.kv2,-1)=COALESCE (col2.kv2,col1.kv2,-1)
 AND COALESCE (col1.kv3,col2.kv3,-1)=COALESCE (col2.kv3,col1.kv3,-1)
 AND COALESCE (col1.kv4,col2.kv4,-1)=COALESCE (col2.kv4,col1.kv4,-1)
FULL OUTER JOIN (SELECT * FROM ContextualValues WHERE Tab=1001 AND Col=1103)
  AS col3
  ON COALESCE (col1.RowID,col2.RowID)=col3.RowID
 AND COALESCE (col1.kv1,col2.kv1,col3.kv1,-1)=COALESCE (col3.kv1,col1.kv1,col2.kv1,-1)
 AND COALESCE (col1.kv2,col2.kv2,col3.kv2,-1)=COALESCE (col3.kv2,col1.kv2,col2.kv2,-1)
 AND COALESCE (col1.kv3,col2.kv3,col3.kv3,-1)=COALESCE (col3.kv3,col1.kv3,col2.kv3,-1)
 AND COALESCE (col1.kv4,col2.kv4,col3.kv4,-1)=COALESCE (col3.kv4,col1.kv4,col2.kv4,-1)
FULL OUTER JOIN (SELECT * FROM ContextualValues WHERE Tab=1001 AND Col=1104)
  AS col4
  ON COALESCE (col1.RowID,col2.RowID,col3.RowID)=col4.RowID
 AND COALESCE (col1.kv1,col2.kv1,col3.kv1,col4.kv1,-1)
    = COALESCE (col4.kv1,col1.kv1,col2.kv1,col3.kv1,-1)
 AND COALESCE (col1.kv2,col2.kv2,col3.kv2,col4.kv2,-1)
    = COALESCE (col4.kv2,col1.kv2,col2.kv2,col3.kv2,-1)
 AND COALESCE (col1.kv3,col2.kv3,col3.kv3,col4.kv3,-1)
    = COALESCE (col4.kv3,col1.kv3,col2.kv3,col3.kv3,-1)
 AND COALESCE (col1.kv4,col2.kv4,col3.kv4,col4.kv4,-1)
    = COALESCE (col4.kv4,col1.kv4,col2.kv4,col3.kv4,-1)
FULL OUTER JOIN (SELECT * FROM ContextualValues WHERE Tab=1001 AND Col=1105)
  AS col5
  ON COALESCE (col1.RowID,col2.RowID,col3.RowID,col4.RowID)=col5.RowID
 AND COALESCE (col1.kv1,col2.kv1,col3.kv1,col4.kv1,col5.kv1,-1)
```

```
    = COALESCE (col5.kv1,col1.kv1,col2.kv1,col3.kv1,col4.kv1,-1)
 AND COALESCE (col1.kv2,col2.kv2,col3.kv2,col4.kv2,col5.kv2,-1)
    = COALESCE (col5.kv2,col1.kv2,col2.kv2,col3.kv2,col4.kv2,-1)
 AND COALESCE (col1.kv3,col2.kv3,col3.kv3,col4.kv3,col5.kv3,-1)
    = COALESCE (col5.kv3,col1.kv3,col2.kv3,col3.kv3,col4.kv3,-1)
 AND COALESCE (col1.kv4,col2.kv4,col3.kv4,col4.kv4,col5.kv4,-1)
    = COALESCE (col5.kv4,col1.kv4,col2.kv4,col3.kv4,col4.kv4,-1)
```

Listing C.6: SQL-View V_MODULE_DERIVED für abgeleitete Tupel

```
SELECT Language, Region, Version, Pricegrade,
       Name, Price, Description, Norm, Material
  FROM V_Module
WHERE RowID = $r$
```

Listing C.7: Umsetzung der Anfrage `IS**01` mittels gSQL

```
SELECT Language, Region, Version, Pricegrade,
       Name, Price, Description, Norm, Material
  FROM V_Module
WHERE RowID = $r$
  AND (Language IN (207, 208, 209) OR Region = 307)
```

Listing C.8: Umsetzung der Anfrage `IS+*02` mittels gSQL

```
CREATE VARIABLE lastID INT;

INSERT INTO AspectValue (Tab,Col,RowID,Value) VALUES (1001, 1101, $r$, 'screw');
SET lastID TO IDENTITY_VAL_LOCAL();
INSERT INTO AspectAssign (Aspect,KeyValue,AspectValue) VALUES (101,214,lastID);

INSERT INTO AspectValue (Tab,Col,RowID,Value) VALUES (1001, 1103, $r$, 'hex socket');
SET lastID TO IDENTITY_VAL_LOCAL();
INSERT INTO AspectAssign (Aspect,KeyValue,AspectValue) VALUES (101,214,lastID);

INSERT INTO AspectValue (Tab,Col,RowID,Value) VALUES (1001, 1105, $r$, 'stainless steel');
SET lastID TO IDENTITY_VAL_LOCAL();
INSERT INTO AspectAssign (Aspect,KeyValue,AspectValue) VALUES (101,214,lastID);

INSERT INTO AspectValue (Tab,Col,RowID,Value) VALUES (1001, 1102, $r$, '0.33');
SET lastID TO IDENTITY_VAL_LOCAL();
INSERT INTO AspectAssign (Aspect,KeyValue,AspectValue)
VALUES (102,310,lastID), (104,501,lastID);

DROP VARIABLE lastID;
```

Listing C.9: Umsetzung der Anfrage `II1103` mittels gSQL

```
DELETE
  FROM AspectValue
 WHERE AspValID IN
       (SELECT ValID
          FROM ContextualValues
         WHERE Tab=1001 AND RowID=$r$
           AND kv2=302 AND kv3=403 AND kv4=501)
```

Listing C.10: Umsetzung der Anfrage `ID1*04` mittels gSQL

```
DECLARE GLOBAL TEMPORARY TABLE Temp (ValID int, Value varchar(255));
CREATE VARIABLE lastID INT;

INSERT INTO Temp
    (SELECT ValID, Value
       FROM ContextualValues
      WHERE Tab=1001 AND Col=1102 AND RowID=$r$
        AND kv2=302 AND kv4=501 AND kv3=-1);

INSERT INTO AspectValue (Tab, Col, RowID, Value)
    (SELECT 1001, 1102, $r$, Value
       FROM Temp);

SET lastID TO IDENTITY_VAL_LOCAL();
INSERT INTO AspectAssign (Aspect, KeyValue, AspectValue)
VALUES (102, 302, lastID), (103, 415, lastID), (104, 501, lastID);

UPDATE AspectValue
   SET Value='0.44'
 WHERE AspValID IN (SELECT ValID FROM TEMP);

DROP VARIABLE lastID;
DROP TABLE TEMP;
```

Listing C.11: Umsetzung der Anfrage `IU1105` mittels gSQL

```
SELECT Language, Region, Version, Pricegrade,
       Name, Price, Description, Norm, Material
 FROM V_Module
```

Listing C.12: Umsetzung der Anfrage `MS**06` mittels gSQL

```
SELECT RowID,
       Language, Region, Version,
       Name, Price, Description
  FROM V_Module_Derived
 WHERE Price>=0.5 AND Price<=1.0
   AND Pricegrade = 501
   AND Region IN (302, 304, 306, 307, 308)
```

Listing C.13: Umsetzung der Anfrage `MS++07` mittels gSQL

```
DELETE FROM AspectValue
 WHERE Table=1001
```

Listing C.14: Umsetzung der Anfrage `MD**08` mittels gSQL

```
DELETE FROM AspectValue
 WHERE AspValID IN
    (SELECT AspectValue
       FROM AspectAssign
      WHERE Aspect=103 AND KeyValue IN (401,402))
```

Listing C.15: Umsetzung der Anfrage `MD++09` mittels gSQL

C.3 Workload-Realisierung mit sSQL

```
WITH sp AS
(SELECT
      COALESCE (av1.Col, av2.Col, av3.Col, av4.Col) AS Col,
      COALESCE (av1.AspValID, av2.AspValID, av3.AspValID, av4.AspValID) AS AspValID,
      COALESCE (av1.Value, av2.Value, av3.Value, av4.Value) AS Value,
      COALESCE (aa1.KeyValue,-1) kv1,
      COALESCE (aa2.KeyValue,-1) kv2,
      COALESCE (aa3.KeyValue,-1) kv3,
      COALESCE (aa4.KeyValue,-1) kv4
   FROM (AspectValue av1 JOIN AspectAssign aa1 ON av1.RowID=$r$ AND
               av1.Tab=1001 AND aa1.Aspect=101 AND aa1.AspectValue=av1.AspValID)
 FULL OUTER JOIN (AspectValue av2 JOIN AspectAssign aa2 ON av2.RowID=$r$ AND
               av2.Tab=1001 AND aa2.Aspect=102 AND aa2.AspectValue=av2.AspValID)
      ON av1.Col=av2.Col
     AND av1.AspValID=av2.AspValID
 FULL OUTER JOIN (AspectValue av3 JOIN AspectAssign aa3 ON av3.RowID=$r$ AND
               av3.Tab=1001 AND aa3.Aspect=103 AND aa3.AspectValue=av3.AspValID)
      ON COALESCE (av1.Col, av2.Col)=av3.Col
     AND COALESCE (av1.AspValID, av2.AspValID)=av3.AspValID
 FULL OUTER JOIN (AspectValue av4 JOIN AspectAssign aa4 ON av4.RowID=$r$ AND
               av4.Tab=1001 AND aa4.aspect=104 AND aa4.AspectValue=av4.AspValID)
      ON COALESCE (av1.Col, av2.Col, av3.Col)=av4.Col
     AND COALESCE (av1.AspValID, av2.AspValID, av4.AspValID)=av4.AspValID
),
sp1 AS (SELECT * FROM sp WHERE Col=1101),
sp2 AS (SELECT * FROM sp WHERE Col=1102),
sp3 AS (SELECT * FROM sp WHERE Col=1103),
sp4 AS (SELECT * FROM sp WHERE Col=1104),
sp5 AS (SELECT * FROM sp WHERE Col=1105)
SELECT
     COALESCE (sp1.kv1, sp2.kv1, sp3.kv1, sp4.kv1) AS Language,
     COALESCE (sp1.kv2, sp2.kv2, sp3.kv2, sp4.kv2) AS Region,
     COALESCE (sp1.kv3, sp2.kv3, sp3.kv3, sp4.kv3) AS Version,
     COALESCE (sp1.kv4, sp2.kv4, sp3.kv4, sp4.kv4) AS Pricegrade,
     sp1.Value AS Name,
     sp2.Value AS Price,
     sp3.Value AS Description,
     sp4.Value AS Norm,
     sp5.Value AS Material
   FROM sp1
FULL OUTER JOIN sp2
      ON sp1.kv1=sp2.kv1
     AND sp1.kv2=sp2.kv2
     AND sp1.kv3=sp2.kv3
     AND sp1.kv4=sp2.kv4
FULL OUTER JOIN sp3
      ON COALESCE (sp1.kv1, sp2.kv1)=sp3.kv1
     AND COALESCE (sp1.kv2, sp2.kv2)=sp3.kv2
     AND COALESCE (sp1.kv3, sp2.kv3)=sp3.kv3
     AND COALESCE (sp1.kv4, sp2.kv4)=sp3.kv4
FULL OUTER JOIN sp4
      ON COALESCE (sp1.kv1, sp2.kv1, sp3.kv1)=sp4.kv1
     AND COALESCE (sp1.kv2, sp2.kv2, sp3.kv2)=sp4.kv2
     AND COALESCE (sp1.kv3, sp2.kv3, sp3.kv3)=sp4.kv3
```

```
    AND COALESCE (sp1.kv4, sp2.kv4, sp3.kv4)=sp4.kv4
FULL OUTER JOIN sp5
     ON COALESCE (sp1.kv1, sp2.kv1, sp3.kv1, sp4.kv1)=sp5.kv1
    AND COALESCE (sp1.kv2, sp2.kv2, sp3.kv2, sp4.kv2)=sp5.kv2
    AND COALESCE (sp1.kv3, sp2.kv3, sp3.kv3, sp4.kv3)=sp5.kv3
    AND COALESCE (sp1.kv4, sp2.kv4, sp3.kv4, sp4.kv4)=sp5.kv4
```

Listing C.16: Umsetzung der Anfrage `IS**01` mittels sSQL

```
WITH sp AS
(SELECT
      COALESCE (av1.Col, av2.Col, av3.Col, av4.Col) AS Col,
      COALESCE (av1.AspValID, av2.AspValID, av3.AspValID, av4.AspValID) AS AspValID,
      COALESCE (av1.Value, av2.Value, av3.Value, av4.Value) AS Value,
      COALESCE (aa1.KeyValue,-1) kv1,
      COALESCE (aa2.KeyValue,-1) kv2,
      COALESCE (aa3.KeyValue,-1) kv3,
      COALESCE (aa4.KeyValue,-1) kv4
   FROM (AspectValue av1 JOIN AspectAssign aa1 ON av1.RowID=$r$ AND
                av1.Tab=1001 AND aa1.Aspect=101 AND aa1.AspectValue=av1.AspValID)
 FULL OUTER JOIN (AspectValue av2 JOIN AspectAssign aa2 ON av2.RowID=$r$ AND
                av2.Tab=1001 AND aa2.Aspect=102 AND aa2.AspectValue=av2.AspValID)
     ON av1.Col=av2.Col
    AND av1.AspValID=av2.AspValID
 FULL OUTER JOIN (AspectValue av3 JOIN AspectAssign aa3 ON av3.RowID=$r$ AND
                av3.Tab=1001 AND aa3.Aspect=103 AND aa3.AspectValue=av3.AspValID)
     ON COALESCE (av1.Col, av2.Col)=av3.Col
    AND COALESCE (av1.AspValID, av2.AspValID)=av3.AspValID
 FULL OUTER JOIN (AspectValue av4 JOIN AspectAssign aa4 ON av4.RowID=$r$ AND
                av4.Tab=1001 AND aa4.aspect=104 AND aa4.AspectValue=av4.AspValID)
     ON COALESCE (av1.Col, av2.Col, av3.Col)=av4.Col
    AND COALESCE (av1.AspValID, av2.AspValID, av4.AspValID)=av4.AspValID
 WHERE aa1.KeyValue IN (207, 208, 209)
       OR aa2.KeyValue=307
),
sp1 AS (SELECT * FROM sp WHERE Col=1101),
sp2 AS (SELECT * FROM sp WHERE Col=1102),
sp3 AS (SELECT * FROM sp WHERE Col=1103),
sp4 AS (SELECT * FROM sp WHERE Col=1104),
sp5 AS (SELECT * FROM sp WHERE Col=1105)
SELECT
     COALESCE (sp1.kv1, sp2.kv1, sp3.kv1, sp4.kv1) AS Language,
     COALESCE (sp1.kv2, sp2.kv2, sp3.kv2, sp4.kv2) AS Region,
     COALESCE (sp1.kv3, sp2.kv3, sp3.kv3, sp4.kv3) AS Version,
     COALESCE (sp1.kv4, sp2.kv4, sp3.kv4, sp4.kv4) AS Pricegrade,
     sp1.Value AS Name,
     sp2.Value AS Price,
     sp3.Value AS Description,
     sp4.Value AS Norm,
     sp5.Value AS Material
  FROM sp1
FULL OUTER JOIN sp2
     ON sp1.kv1=sp2.kv1
    AND sp1.kv2=sp2.kv2
    AND sp1.kv3=sp2.kv3
    AND sp1.kv4=sp2.kv4
FULL OUTER JOIN sp3
```

```
     ON COALESCE (sp1.kv1, sp2.kv1)=sp3.kv1
    AND COALESCE (sp1.kv2, sp2.kv2)=sp3.kv2
    AND COALESCE (sp1.kv3, sp2.kv3)=sp3.kv3
    AND COALESCE (sp1.kv4, sp2.kv4)=sp3.kv4
FULL OUTER JOIN sp4
     ON COALESCE (sp1.kv1, sp2.kv1, sp3.kv1)=sp4.kv1
    AND COALESCE (sp1.kv2, sp2.kv2, sp3.kv2)=sp4.kv2
    AND COALESCE (sp1.kv3, sp2.kv3, sp3.kv3)=sp4.kv3
    AND COALESCE (sp1.kv4, sp2.kv4, sp3.kv4)=sp4.kv4
FULL OUTER JOIN sp5
     ON COALESCE (sp1.kv1, sp2.kv1, sp3.kv1, sp4.kv1)=sp5.kv1
    AND COALESCE (sp1.kv2, sp2.kv2, sp3.kv2, sp4.kv2)=sp5.kv2
    AND COALESCE (sp1.kv3, sp2.kv3, sp3.kv3, sp4.kv3)=sp5.kv3
    AND COALESCE (sp1.kv4, sp2.kv4, sp3.kv4, sp4.kv4)=sp5.kv4
```

Listing C.17: Umsetzung der Anfrage `IS+*02` mittels sSQL

```
CREATE VARIABLE lastID INT;

INSERT INTO AspectValue (Tab,Col,RowID,Value) VALUES (1001, 1101, $r$, 'screw');
SET lastID TO IDENTITY_VAL_LOCAL();
INSERT INTO AspectAssign (Aspect,KeyValue,AspectValue) VALUES (101,214,lastID);

INSERT INTO AspectValue (Tab,Col,RowID,Value) VALUES (1001, 1103, $r$, 'hex socket');
SET lastID TO IDENTITY_VAL_LOCAL();
INSERT INTO AspectAssign (Aspect,KeyValue,AspectValue) VALUES (101,214,lastID);

INSERT INTO AspectValue (Tab,Col,RowID,Value) VALUES (1001, 1105, $r$, 'stainless steel');
SET lastID TO IDENTITY_VAL_LOCAL();
INSERT INTO AspectAssign (Aspect,KeyValue,AspectValue) VALUES (101,214,lastID);

INSERT INTO AspectValue (Tab,Col,RowID,Value) VALUES (1001, 1102, $r$, '0.33');
SET lastID TO IDENTITY_VAL_LOCAL();
INSERT INTO AspectAssign (Aspect,KeyValue,AspectValue)
VALUES (102,310,lastID), (104,501,lastID);

DROP VARIABLE lastID;
```

Listing C.18: Umsetzung der Anfrage `II1103` mittels sSQL

```
DELETE FROM AspectValue
 WHERE AspValID IN
       (SELECT COALESCE (av1.AspValID, av2.AspValID, av3.AspValID) AS AspValID,
          FROM (AspectValue av1 JOIN AspectAssign aa1 ON av1.RowID=$r$ AND
                av1.Tab=1001 AND aa1.Aspect=102 AND aa1.AspectValue=av1.AspValID)
          JOIN (AspectValue av2 JOIN AspectAssign aa2 ON av2.RowID=$r$ AND
                av2.Tab=1001 AND aa2.Aspect=103 AND aa2.AspectValue=av2.AspValID)
            ON av1.Col=av2.Col AND av1.AspValID=av2.AspValID AND av1.RowID=av2.RowID
          JOIN (AspectValue av3 JOIN AspectAssign aa3 ON av3.RowID=$r$ AND
               av3.Tab=1001 AND aa3.Aspect=104 AND aa3.AspectValue=av3.AspValID)
            ON COALESCE (av1.Col, av2.Col)=av3.Col
           AND COALESCE (av1.AspValID, av2.AspValID)=av3.AspValID
           AND COALESCE (av1.RowID, av2.RowID)=av3.RowID
         WHERE aa1.KeyValue=302 AND aa3.KeyValue=501 AND aa2.KeyValue=403
       )
```

Listing C.19: Umsetzung der Anfrage `ID1*04` mittels sSQL

```
DECLARE GLOBAL TEMPORARY TABLE Temp (ValID int, Value varchar(255));
CREATE VARIABLE lastID INT;

INSERT INTO Temp
  (SELECT av1.AspValID as ValID, av1.Value AS Value
     FROM (AspectValue av1 JOIN AspectAssign aa1 ON av1.RowID=$r$ AND av1.Tab=1001
           AND av1.Col=1102 AND aa1.Aspect=102 AND aa1.AspectValue=av1.AspValID)
     JOIN (AspectValue av2 JOIN AspectAssign aa2 ON av2.RowID=$r$ AND av2.Tab=1001
           AND av2.Col=1102 AND aa2.Aspect=104 AND aa2.AspectValue=av2.AspValID)
       ON av1.AspValID=av2.AspValID
    WHERE aa1.KeyValue=302 AND aa2.KeyValue=501
  )

INSERT INTO AspectValue (Tab, Col, RowID, Value)
    (SELECT 1001, 1102, $r$, Value
       FROM Temp);

SET lastID TO IDENTITY_VAL_LOCAL();
INSERT INTO AspectAssign (Aspect, KeyValue, AspectValue)
VALUES (102, 302, lastID), (103, 415, lastID), (104, 501, lastID);

UPDATE AspectValue
   SET Value='0.44'
 WHERE AspValID IN (SELECT ValID FROM TEMP);

DROP VARIABLE lastID;
DROP TABLE TEMP;
```

Listing C.20: Umsetzung der Anfrage `IU1105` mittels sSQL

```
WITH sp AS
(SELECT
      COALESCE (av1.RowID, av2.RowID, av3.RowID, av4.RowID) AS RowID,
      COALESCE (av1.Col, av2.Col, av3.Col, av4.Col) AS Col,
      COALESCE (av1.AspValID, av2.AspValID, av3.AspValID, av4.AspValID) AS AspValID,
      COALESCE (av1.Value, av2.Value, av3.Value, av4.Value) AS Value,
      aa1.AspAssID AssID1, COALESCE (aa1.KeyValue,-1) kv1,
      aa2.AspAssID AssID2, COALESCE (aa2.KeyValue,-1) kv2,
      aa3.AspAssID AssID3, COALESCE (aa3.KeyValue,-1) kv3,
      aa4.AspAssID AssID4, COALESCE (aa4.KeyValue,-1) kv4
   FROM (AspectValue av1 JOIN AspectAssign aa1 ON av1.Tab=1001 AND
                  aa1.Aspect=101 AND aa1.AspectValue=av1.AspValID)
 FULL OUTER JOIN (AspectValue av2 JOIN AspectAssign aa2 ON av2.Tab=1001 AND
                  aa2.Aspect=102 AND aa2.AspectValue=av2.AspValID)
     ON av1.RowID=av2.RowID
    AND av1.Col=av2.Col
    AND av1.AspValID=av2.AspValID
 FULL OUTER JOIN (AspectValue av3 JOIN AspectAssign aa3 ON av3.Tab=1001 AND
                  aa3.Aspect=103 AND aa3.AspectValue=av3.AspValID)
     ON COALESCE (av1.RowID, av2.RowID)=av3.RowID
    AND COALESCE (av1.Col, av2.Col)=av3.Col
    AND COALESCE (av1.AspValID, av2.AspValID)=av3.AspValID
 FULL OUTER JOIN (AspectValue av4 JOIN AspectAssign aa4 ON av4.Tab=1001 AND
                  aa4.Aspect=104 AND aa4.AspectValue=av4.AspValID)
     ON COALESCE (av1.RowID, av2.RowID, av3.RowID)=av4.RowID
    AND COALESCE (av1.Col, av2.Col, av3.Col)=av4.Col
```

```
    AND COALESCE (av1.AspValID, av2.AspValID, av4.AspValID)=av4.AspValID),
sp1 AS (SELECT * FROM sp WHERE Col=1101),
sp2 AS (SELECT * FROM sp WHERE Col=1102),
sp3 AS (SELECT * FROM sp WHERE Col=1103),
sp4 AS (SELECT * FROM sp WHERE Col=1104),
sp5 AS (SELECT * FROM sp WHERE Col=1105)
SELECT
    COALESCE (sp1.RowID, sp2.RowID, sp3.RowID, sp4.RowID) AS RowID,
    COALESCE (sp1.kv1, sp2.kv1, sp3.kv1, sp4.kv1) AS Language,
    COALESCE (sp1.kv2, sp2.kv2, sp3.kv2, sp4.kv2) AS Region,
    COALESCE (sp1.kv3, sp2.kv3, sp3.kv3, sp4.kv3) AS Version,
    COALESCE (sp1.kv4, sp2.kv4, sp3.kv4, sp4.kv4) AS Pricegrade,
    sp1.Value AS Name,
    sp2.Value AS Price,
    sp3.Value AS Description,
    sp4.Value AS Norm,
    sp5.Value AS Material
  FROM sp1
FULL OUTER JOIN sp2
     ON sp1.RowID=sp2.RowID
    AND sp1.kv1=sp2.kv1
    AND sp1.kv2=sp2.kv2
    AND sp1.kv3=sp2.kv3
    AND sp1.kv4=sp2.kv4
FULL OUTER JOIN sp3
     ON COALESCE (sp1.RowID, sp2.RowID)=sp3.RowID
    AND COALESCE (sp1.kv1, sp2.kv1)=sp3.kv1
    AND COALESCE (sp1.kv2, sp2.kv2)=sp3.kv2
    AND COALESCE (sp1.kv3, sp2.kv3)=sp3.kv3
    AND COALESCE (sp1.kv4, sp2.kv4)=sp3.kv4
FULL OUTER JOIN sp4
     ON COALESCE (sp1.RowID,sp2.RowID,sp3.RowID)=sp4.RowID
    AND COALESCE (sp1.kv1, sp2.kv1, sp3.kv1)=sp4.kv1
    AND COALESCE (sp1.kv2, sp2.kv2, sp3.kv2)=sp4.kv2
    AND COALESCE (sp1.kv3, sp2.kv3, sp3.kv3)=sp4.kv3
    AND COALESCE (sp1.kv4, sp2.kv4, sp3.kv4)=sp4.kv4
FULL OUTER JOIN sp5
     ON COALESCE (sp1.RowID,sp2.RowID,sp3.RowID,sp4.RowID)=sp5.RowID
    AND COALESCE (sp1.kv1, sp2.kv1, sp3.kv1, sp4.kv1)=sp5.kv1
    AND COALESCE (sp1.kv2, sp2.kv2, sp3.kv2, sp4.kv2)=sp5.kv2
    AND COALESCE (sp1.kv3, sp2.kv3, sp3.kv3, sp4.kv3)=sp5.kv3
    AND COALESCE (sp1.kv4, sp2.kv4, sp3.kv4, sp4.kv4)=sp5.kv4
```

Listing C.21: Umsetzung der Anfrage `MS**06` mittels sSQL

```
WITH sp AS
(SELECT
     COALESCE (av1.RowID, av2.RowID, av3.RowID, av4.RowID) AS RowID,
     COALESCE (av1.Col, av2.Col, av3.Col, av4.Col) AS Col,
     COALESCE (av1.AspValID, av2.AspValID, av3.AspValID, av4.AspValID) AS AspValID,
     COALESCE (av1.Value, av2.Value, av3.Value, av4.Value) AS Value,
     aa1.KeyValue kv1,
     aa2.KeyValue kv2,
     aa3.KeyValue kv3,
     aa4.KeyValue kv4
   FROM (AspectValue av1 JOIN AspectAssign aa1 ON av1.Tab=1001 AND
                 aa1.Aspect=101 AND aa1.AspectValue=av1.AspValID)
```

```
 FULL OUTER JOIN (AspectValue av2 JOIN AspectAssign aa2 ON av2.Tab=1001 AND
                  aa2.Aspect=102 AND aa2.AspectValue=av2.AspValID AND
                  aa2.KeyValue IN (302, 304, 306, 307, 308))
     ON av1.RowID=av2.RowID
    AND av1.Col=av2.Col
    AND av1.AspValID=av2.AspValID
 FULL OUTER JOIN (AspectValue av3 JOIN AspectAssign aa3 ON av3.Tab=1001 AND
                  aa3.Aspect=103 AND aa3.AspectValue=av3.AspValID)
     ON COALESCE (av1.RowID, av2.RowID)=av3.RowID
    AND COALESCE (av1.Col, av2.Col)=av3.Col
    AND COALESCE (av1.AspValID, av2.AspValID)=av3.AspValID
 FULL OUTER JOIN (AspectValue av4 JOIN AspectAssign aa4 ON av4.Tab=1001 AND
                  aa4.Aspect=104 AND aa4.AspectValue=av4.AspValID AND
                  aa4.KeyValue=501)
     ON COALESCE (av1.RowID, av2.RowID, av3.RowID)=av4.RowID
    AND COALESCE (av1.Col, av2.Col, av3.Col)=av4.Col
    AND COALESCE (av1.AspValID, av2.AspValID, av4.AspValID)=av4.AspValID),
sp1 AS (SELECT * FROM sp WHERE Col=1101
                           AND kv2 IN (302, 304, 306, 307, 308) AND kv4 = 501),
sp2 AS (SELECT * FROM sp WHERE Col=1102 AND kv4 = 501
                           AND CAST(Value AS NUMERIC) >= 0.5
                           AND CAST(Value AS NUMERIC) <= 1.0),
sp3 AS (SELECT * FROM sp WHERE Col=1103
                           AND kv2 IN (302, 304, 306, 307, 308) AND kv4 = 501)
SELECT DISTINCT
    COALESCE (sp1.RowID, sp2.RowID, sp3.RowID) RowID,
    COALESCE (sp1.kv1, sp2.kv1, sp3.kv1) kv1,
    COALESCE (sp1.kv2, sp2.kv2, sp3.kv2) kv2,
    COALESCE (sp1.kv3, sp2.kv3, sp3.kv3) kv3,
    COALESCE (sp1.kv4, sp2.kv4, sp3.kv4) kv4,
    sp1.AspValID, sp2.AspValID, sp3.AspValID
  FROM sp2
LEFT OUTER JOIN sp1 ON sp1.RowID=sp2.RowID
   AND COALESCE (sp1.kv1, sp2.kv1)=COALESCE (sp2.kv1, sp1.kv1)
   AND COALESCE (sp1.kv2, sp2.kv2)=COALESCE (sp2.kv2, sp1.kv2)
   AND COALESCE (sp1.kv3, sp2.kv3)=COALESCE (sp2.kv3, sp1.kv3)
   AND COALESCE (sp1.kv4, sp2.kv4)=COALESCE (sp2.kv4, sp1.kv4)
LEFT OUTER JOIN sp3 ON COALESCE (sp1.RowID, sp2.RowID)=sp3.RowID
   AND COALESCE (sp1.kv1, sp2.kv1, sp3.kv1)=COALESCE (sp3.kv1, sp1.kv1, sp2.kv1)
   AND COALESCE (sp1.kv2, sp2.kv2, sp3.kv2)=COALESCE (sp3.kv2, sp1.kv2, sp2.kv2)
   AND COALESCE (sp1.kv3, sp2.kv3, sp3.kv3)=COALESCE (sp3.kv3, sp1.kv3, sp2.kv3)
   AND COALESCE (sp1.kv4, sp2.kv4, sp3.kv4)=COALESCE (sp3.kv4, sp1.kv4, sp2.kv4)
```

Listing C.22: Umsetzung der Anfrage `MS++07` mittels sSQL

```
DELETE FROM AspectValue
 WHERE Table=1001
```

Listing C.23: Umsetzung der Anfrage `MD**08` mittels sSQL

```
DELETE FROM AspectValue
 WHERE AspValID IN
    (SELECT AspectValue
       FROM AspectAssign
      WHERE Aspect=103 AND KeyValue IN (401,402))
```

Listing C.24: Umsetzung der Anfrage `MD+*09` mittels sSQL

Literaturverzeichnis

[Abe04] Helmut Abels. Organisationsformen der Produktion. Vorlesung Produktionsplanung und -steuerung I+II, Fachhochschule Köln, Fakultät für Fahrzeugsysteme und Produktion, 2004. → Seite 10, 11, 193

[AGJ+08] Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper und Jan Rittinger. Multi-tenant databases for software as a service: schema-mapping techniques. In *SIGMOD Conference*, Seiten 1195–1206, 2008. → Seite 52, 96

[Aka09] Akamai Technolgies Inc. *Akamai bestätigt Site-Performance als bestimmenden Faktor für die Kundenloyalität im Internet.* Pressemitteilung, Akamai Technolgies Inc., 2009. → Seite 16

[ALB10] Alexander Adam, Sebastian Leuoth und Wolfgang Benn. Nutzung von Proxys zur Ergänzung von Datenbankfunktionen. In *Proceedings of the 22nd GI Workshop Grundlagen von Datenbanken 2010*, Seiten 31–35, Bad Helmstedt, Germany, Mai 2010. → Seite 96, 98

[All06] J. Allen. *The Unicode Standard, Version 5.0.* Addison-Wesley Professional, 2006. → Seite 43

[AMD07] Advanced Micro Devices. *AMD Athlon 64 X2 Dual-Core Processor Product Data Sheet*, Januar 2007. Revision 3.10. → Seite 133

[ASA+09] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens und Nina Amenta. Real-time Parallel Hashing on the GPU. In *Proceedings of ACM SIGGRAPH Asia 2009*, Seiten 154:1–154:9, Yokohama, Japan, 2009. → Seite 158

[BAKF04] Thorsten Blecker, Nizar Abdelkafia, Gerold Kreuter und Gerhard Friedrich. Product Configuration Systems: State-of-the-Art, Conceptualization and Extensions. In *Proceedings of the 8th Maghrebian Conference on Software Engineering and Artificial Intelligence (MCSEAI'04)*, Seiten 25–36, Sousse, Tunisia, Mai 2004. → Seite 15, 30

[Bar94] Martin Bartuschat. *Beitrag zur Beherrschung der Variantenvielfalt in der Serienfertigung.* Dissertation, TU Braunschweig, 1994. → Seite 8

[BCTV04] Deepavali Bhagwat, Laura Chiticariu, Wang-Chiew Tan und Gaurav Vijayvargiya. An Annotation Management System For Relational Databases. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB'04)*, Seiten 900–911, Toronto, Canada, 2004. → Seite 69

[Bec83] Ulrich Beck. Jenseits von Stand und Klasse? Soziale Ungleichheiten, gesellschaftliche Individualisierungsprozesse und die Entstehung neuer sozialer Formationen und Identitäten. In *Soziale Ungleichheiten. Soziale Welt*, Seiten 35–74. Reinhard Kreckel, Göttingen, 1983. → Seite 1

[BG08] Andreas Bauer und Holger Günzel. *Data-Warehouse-Systeme: Architektur, Entwicklung, Anwendung.* dpunkt Verlag, 2008. → Seite 45

[BGHK01] Karl-Werner Brand, Robert Gugutzer, Angelika Heimerl und Alexander Kupfahl. Sozialwissenschaftliche Analysen zu Veränderungsmöglichkeiten nachhaltiger Konsummuster (Zsfg.). Forschungsbericht 20017155, UNESCO-Verbindungsstelle im Umweltbundesamt, Berlin, 2001. → Seite 1

[BI97] Jürgen Bloech und Gösta B. Ihde (Hrsg.). *Vahlens Großes Logistiklexikon. Logistik total.* Vahlen, 1997. → Seite 8

[Bie01] Christian Bieniek. *Prozeßorientierte Produktkonfiguration zur integrierten Auftragsabwicklung bei Variantenfertigern.* Dissertation, Technische Universität Carolo-Wilhelmina zu Braunschweig, gem. Fakultät für Maschinenbau und Elektrotechnik, Juli 2001. → Seite 8

[Bri99] Axel Brinkop. *Variantenkonstruktion durch Auswertung der Abhängigkeiten zwischen den Konstruktionsbauteilen.* Infix, 1999. → Seite 159

[Bru09] Erwin Brunner. Jakob der Reiche. Wie vor 500 Jahren ein Kaufmann aus Augsburg die Globalisierung erfand. *National Geographic Deutschland*, 51(3):27–51, März 2009. → Seite 2

[Cat10] Rick Cattell. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 39(4):12–27, Dezember 2010. → Seite 82

[CB74] Donald D. Chamberlin und Raymond F. Boyce. SEQUEL: A structured English query language. In *SIGFIDET '74: Proceedings of the 1974 ACM SIGFIDET workshop on Data description, access and control*, Seiten 249–264, Ann Arbor, Michigan, USA, Mai 1974. → Seite 56

[CGGL04] Conor Cunningham, Goetz Graefe und César A. Galindo-Legaria. PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS. In *(e)Proceedings of the 30th International Conference on Very Large Data Bases*, Seiten 998–1009, 2004. → Seite 92, 94

[CHA09] Raul Chong, Ian Hakes und Rav Ahuja. *Getting Started with DB2 Express-C.* IBM Corporation, 3. Auflage, 2009. → Seite 133

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, 1970. → Seite 51, 56, 57, 87

[Cud06] Brian J. Cudahy. The Containership Revolution: Malcom McLean's 1956 Innovation Goes Global. *TR News*, 246:5–9, 2006. → Seite 2

[DIN03] DIN Deutsches Institut für Normung e.V. (Hrsg.). *Dokumentationswesen.* Berlin: Beuth, 2003. → Seite 8, 13

[Dit11] **Maik Dittrich. Prototypische Implementierung eines Tools zur Integration Funktionaler Aspekte in bestehende relationale Datenbankmodelle. Bachelorarbeit, Berufsakademie Gera, Juli 2011.** → Seite 55

[DNB06] Valentin Dinu, Prakash Nadkarni und Cynthia Brandt. Pivoting approaches for bulk extraction of Entity-Attribute-Value data. *Computer Methods And Programs in Biomedicine*, 82(1):38–43, 2006. → Seite 93, 98

[Dre08] Markus Drews. Interaction Patterns für Produktkonfiguratoren. In *Mensch & Computer 2008: Viel Mehr Interaktion*, Seiten 367–376, Lübeck, September 2008. → Seite 8, 15, 17, 19

[DS05] Wolfgang Domschke und Armin Scholl. *Grundlagen der Betriebswirtschaftslehre. Eine Einführung aus entscheidungsorientierter Sicht.* Springer, 2005. → Seite 9, 191

[Dyr11] Curtis E. Dyreson. Aspect-Oriented Relational Algebra. In *Proceedings of the 14th International Conference on Extending Database Technology (EDBT'11)*, Seiten 377–388, Uppsala, Sweden, März 2011. → Seite 69

[Egy01] T. Egyedi. Why Java Was Not Standardized Twice. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, HICSS '01, Seiten 5015–5025, Wailea-Makena, HI, USA, 2001. → Seite 98

[Els03] Georg Elsner. Kundenbindung durch Produktkonfiguratoren und Baukästen. *Industrie Management*, 13(1):33–36, 2003. → Seite 13, 26

[FHJ+90] Carol Friedman, George Hripcsak, Stephen B. Johnson, James J. Cimino und Paul D. Clayton. A Generalized Relational Schema for an Integrated Clinical Patient Database. In *Proceedings of the 14th Annual Symposium on Computer Applications in Medical Care*, Seiten 335–339, Washington DC, November 1990. → Seite 82

[Fra02] Hans-Joachim Franke. *Variantenmanagement in der Einzel- und Kleinserienfertigung.* Fachbuchverlag Leipzig, 2002. → Seite 8

[Ger12] **Martin Gerstmann. Analyse von Persistierungsmöglichkeiten für funktionale Aspekte. Bachelorarbeit, Fakultät für Mathematik und Informatik, Friedrich-Schiller-Universität Jena, Januar 2012.** → Seite 61, 63, 64

[Göb09] **Andreas Göbel. Internationalisierung in Produktkonfiguratoren – Anforderungen und Konzepte für die Datenhaltung. Diplomarbeit, Fakultät für Mathematik und Informatik, Friedrich-Schiller-Universität Jena, März 2009.** → Seite 12, 15, 16, 23, 41, 42, 43, 44, 45, 46, 47, 48, 52, 157, 191, 193

[Gol06] Christoph Gollmick. *Konzept, Realisierung und Anwendung nutzerdefinierter Replikation in mobilen Datenbanksystemen.* Dissertation, Fakultät für Mathematik und Informatik, Friedrich-Schiller-Universität Jena, 2006. → Seite 25

[Grä04] Iris Gräßler. *Kundenindividuelle Massenproduktion: Entwicklung, Vorbereitung der Herstellung, Veränderungsmanagement.* Springer, 2004. → Seite 2

[HBL02] Z. He, D. W. Bustard und X. Liu. Software Internationalisation and Localisation – Practice and Evolution. In *Proceedings of the naugural International Symposium on Principles and Practice of Programming in Java (PPPJ'02)*, Seiten 89–94, Dublin, Ireland, Juni 2002. → Seite 41

[HBM+11] Christian Hartmann, Jürgen Beyer, Karolina Merai, Matthias Tenten und Markus Wolf. Entwicklung des grenzüberschreitenden Warenhandels. Technischer Bericht, Bundeszentrale für politische Bildung, Bonn, 2011. → Seite 2

[HR01] Theo Härder und Erhard Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Springer, 2001. → Seite 25, 49, 57, 102

[IBM12] IBM. *DB2 v9.7 for Linux, Unix and Windows: SQL Reference, Volume 1*, Juli 2012. → Seite 83

[IL00] S. Iyengar. und M. Lepper. When Choice is Demotivating: Can One Desire Too Much of a Good Thing? *Journal of Personality and social Psychology*, 79(6):995–1006, 2000. → Seite 15

[Irr95] Roland Irrgang. *Entscheidungstabellen-Technik: Entscheidungstabellen erstellen und analysieren*. expert, 1995. → Seite 14

[ISO92] International Organization for Standardization (ISO), Genf. *Information technology – Database languages – SQL*, 1992. ISO/IEC 9075:1992. → Seite 55, 93, 95

[ISO02] International Organization for Standardization (ISO), Genf. *Codes for the representation of names of languages – Part 1: Alpha-2 code*, 2002. ISO/IEC 639-1:2002. → Seite 42, 51

[ISO03] International Organization for Standardization (ISO), Genf. *Information technology – Database languages – SQL – Part 14: XML–Related Specifications (SQL/XML)*, 2003. ISO/IEC 9075-14:2003. → Seite 61

[ISO07] International Organization for Standardization (ISO), Genf. *Codes for the representation of names of countries and their subdivisions – Part 1: Country codes*, 2007. ISO/IEC 3166:2007. → Seite 42, 51

[ISO11] International Organization for Standardization (ISO), Genf. *Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation)*, 2011. ISO/IEC 9075-2:2011. → Seite 94

[Jun02] Matthias Junge. *Individualisierung*. Campus Verlag, 2002. → Seite 1

[Jun06] Hans Jung. *Allgemeine Betriebswirtschaftslehre*. Oldenbourg Wissenschaftsverlag, 2006. → Seite 8

[KASW06] Philip Kotler, Gary Armstrong, John Saunders und Veronica Wong. *Grundlagen des Marketing*. Pearson Studium, 2006. → Seite 8

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, John Irwin und Jean-Marc Loingtier. Aspect-Oriented Programming. In *ECOOP*, Seiten 220–242, Juni 1997. → Seite 50, 69

[Knu71] Donald E. Knuth. Top-down syntax analysis. *Acta Informatica*, 1(2):79–110, 1971. → Seite 119

[KR02] Ralph Kimball und Margy Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. John Wiley & Sons, 2002. → Seite 46, 57

[KRA02] Rolf-Dieter Kempis, Jürgen Ringbeck und R. Augustin. *Do IT smart*. Uebereuter Wirtschaftsverlag, 2002. → Seite 159

[Kru10] **Andreas Krug. Entwurf eines integrativen Grundmodells für Produktkonfiguratoren. Diplomarbeit, Fakultät für Mathematik und Informatik, Friedrich-Schiller-Universität Jena, Juni 2010.** → Seite 8, 9, 10, 12, 13, 14, 15, 19, 21, 35, 191

[KS88] Hans-Peter Kriegel und Bernhard Seeger. PLOP-Hashing: A Grid File without Directory. In *Proceedings of the 4th International Conference on Data Engineering*, Seiten 369–376, Los Angeles, California, USA, 1988. → Seite 158

[KS02] Klaus-Peter Kistner und Marion Steven. *Betriebswirtschaftslehre im Grundstudium 1: Produktion, Absatz, Finanzierung*. Physica-Verlag, 2002. → Seite 8

[Lec06] Thomas Leckner. *Kundenkooperation beim Web-basierten Konfigurieren von Produkten*. Eul, 2006. → Seite 30, 32, 34

[Lev83] Theodore Levitt. The Globalization of Markets. *Harvard Business Review*, 61(3):92–102, 1983. → Seite 1, 2

[Lie03] **Matthias Liebisch. Synchronisationskonflikte beim mobilen Datenbankzugriff: Vermeidung, Erkennung und Behandlung. Diplomarbeit, Fakultät für Mathematik und Informatik, Friedrich-Schiller-Universität Jena, Februar 2003.** → Seite 37, 96

[Lie10a] **Matthias Liebisch. Aspektorientierte Datenhaltung – ein Modellierungsparadigma. In *Proceedings of the 22nd GI Workshop Grundlagen von Datenbanken 2010*, Seiten 13–17, Bad Helmstedt, Mai 2010.** → Seite 49, 50, 55

[Lie10b] **Matthias Liebisch. Supporting functional aspects in relational databases. In *Proceedings of the 2nd International Conference on Software Technology and Engineering (ICSTE)*, Seiten 227–231, San Juan, Puerto Rico, Oktober 2010.** → Seite 58, 84

[Lie11a] **Matthias Liebisch. Analyse und Vergleich von Zugriffstechniken für funktionale Aspekte in RDBMS. In *Proceedings zum 23. GI-Workshop Grundlagen von Datenbanken*, Seiten 25–30, Obergurgl, Österreich, Mai 2011.** → Seite 93, 97

[Lie11b] **Matthias Liebisch. Accessing Functional Aspects with Pure SQL – Lessons Learned. In *Proceedings of the 15th East European Conference on Advances in Databases and Information Systems*, Band 2, Seiten 85–94, Vienna, Austria, September 2011.** → Seite 93, 107

[Lie11c] **Matthias Liebisch. Klassifikation von Daten und Prozessen in Produktkonfiguratoren. In *Proceedings der 41. Jahrestagung der Gesellschaft für Informatik e.V. (GI)*, Seite 173, Berlin, Oktober 2011.** → Seite 18, 19, 24, 40

[LKH11] Dortje Löper, Meike Klettke und Andreas Heuer. Das Entity-Attribute-Value-Konzept als Speicherstruktur für die Informationsintegration in der ambulanten Pflege. In *Proceedings der 41. Jahrestagung der Gesellschaft für Informatik e.V. (GI)*, Seiten 1–15, Berlin, Oktober 2011. → Seite 82

[LP10] **Matthias Liebisch und Matthias Plietz. Performance-Analysen für Realisierungsansätze im Kontext der Aspektorientierten Datenhaltung. Technischer Bericht, Fakultät für Mathematik und Informatik, Friedrich-Schiller-Universität Jena, November 2010.** → Seite 93, 107

[Mic12] Microsoft. *Microsoft SQL Server 2012: Transact-SQL DML Reference*, Mai 2012. → Seite 94

[Mül08] Thomas Müller. Verfahren zur Verarbeitung von XML-Werten in SQL-Anfrageergebnissen. Dissertation, Fakultät für Mathematik und Informatik, Friedrich-Schiller-Universität Jena, April 2008. → Seite 61, 66, 96

[NHS84] J. Nievergelt, Hans Hinterberger und Kenneth C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Transactions on Database Systems*, 9(1):38–71, 1984. → Seite 158

[NMC+99] Prakash Nadkarni, Luis Marenco, Roland Chen, Emmanouil Skoufos, Gordon Shepherd und Perry Miller. Organization of Heterogeneous Scientific Data Using the EAV/CR Representation. *Journal of the American Medical Informatics Association*, 6(6):478–493, 1999. → Seite 58, 82, 83

[Ora12] Oracle. *Oracle Database 11g: SQL Language Reference*, Dezember 2012. → Seite 83, 85, 94

[PGD13] The PostgreSQL Global Development Group. *PostgreSQL 9.0.13 Documentation*, 2013. → Seite 133

[Pie11a] **Bernhard Pietsch. Entwurf einer Zugriffsschicht für funktionale Aspekte in DBMS. Studienarbeit, Fakultät für Mathematik und Informatik, Friedrich-Schiller-Universität Jena, März 2011.** → Seite 49, 70, 84, 92, 99

[Pie11b] **Bernhard Pietsch. Aspektorientierte Datenhaltung in relationalen DBMS – Implementierung und Bewertung einer Zugriffsschicht. Diplomarbeit, Fakultät für Mathematik und Informatik, Friedrich-Schiller-Universität Jena, September 2011.** → Seite 53, 70, 92, 99, 103, 105, 109, 113, 121, 133, 134, 137, 145, 169, 192

[Pin93] B. Joseph Pine. *Mass Customization: The New Frontier in Business Competition*. Harvard Business School Press, 1993. → Seite 2

[PKMS03] Frank Piller, Michael Koch, Kathrin Möslein und Petra Schubert. Managing High Variety: How to Overcome the Mass Confusion Phenomenon of Customer

Co-Design. In *Proceedings of the 3rd Conference of the European Academy of Management (EURAM)*, Milano, Italy, April 2003. → Seite 15

[Pli12] Matthias Plietz. Gleichteilestrategie zur Verringerung der Varianz und deren Voraussetzungen aus Sicht der Praxis. Ausarbeitung zum Seminar „Cutting and Packing“, Wirtschaftswissenschaftliche Fakultät, Friedrich-Schiller-Universität Jena, Februar 2012. → Seite 14, 26

[Pol08] Benjamin Polak. *Kundenorientierte Gestaltung von Produktkonfiguratoren.* Dissertation, Universität St. Gallen, Hochschule für Wirtschafts-, Rechts- und Sozialwissenschaften, 2008. → Seite 12, 14, 16, 33, 34, 159

[PQ95] Terrence J. Parr und Russell W. Quong. ANTLR: A predicated-$LL(k)$ parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995. → Seite 119

[Ras04] Awais Rashid. *Aspect-Oriented Database Systems.* Springer, 2004. → Seite 69

[Ree79] Trygve Reenskaug. Thing-Model-View-Editor – an Example from a planning-system. Technischer Bericht, Xerox PARC, Mai 1979. → Seite 36

[Ree03] Trygve Reenskaug. The Model-View-Controller (MVC) – Its Past and Present. Technischer Bericht, University of Oslo, Mai 2003. → Seite 36

[RI09] A.N.M. Bazlur Rashid und M.S. Islam. Role of Materialized View Maintenance with PIVOT and UNPIVOT Operators. In *Proceedings of the 1st IEEE International Advance Computing Conference (IACC 2009)*, Seiten 915–955, März 2009. → Seite 94

[RP03] Timm Rogoll und Frank Piller. *Konfigurationssysteme für Mass Customization und Variantenproduktion (Marktstudie).* ThinkConsult, 2003. → Seite 11, 16

[RP06] Ralf Reichwald und Frank T. Piller. *Interaktive Wertschöpfung. Open Innovation, Individualisierung und neue Formen der Arbeitsteilung.* Gabler, 2006. → Seite 159

[Rus08] Craig Russell. Bridging the object-relational divide. *ACM Queue*, 6(3):16–26, 2008. → Seite 59

[Sch06] Christian Scheer. *Kundenorientierter Produktkonfigurator: Erweiterung des Produktkonfiguratorkonzeptes zur Vermeidung kundeninitiierter Prozessabbrüche bei Präferenzlosigkeit und Sonderwünschen in der Produktspezifikation.* Logos Berlin, 2006. → Seite 10, 12, 13, 15, 16, 22, 23, 30, 33, 159

[Sch11] **Tilmann Schilling. Realisierungskonzepte für die Aspektorientierte Datenhaltung. Studienarbeit, Fakultät für Mathematik und Informatik, Friedrich-Schiller-Universität Jena, April 2011.** → Seite 52, 57, 64, 83

[Ska05] Steffen Skatulla. Speicherung und Indexierung komplexer Objekte in objektrelationalen Datenbank-Management-Systemen. Dissertation, Fakultät für Mathematik und Informatik, Friedrich-Schiller-Universität Jena, Dezember 2005. → Seite 59

[Ste12] Robert Steingräber. Flexibilisierungskonzepte für relationale Datenbankschemata. Studienarbeit, Fakultät für Mathematik und Informatik, Friedrich-Schiller-Universität Jena, Januar 2012. → Seite 82

[Tap96] Don Tapscott. *Die digitale Revolution. Verheißungen einer vernetzten Welt – die Folgen für Wirtschaft, Management und Gesellschaft.* Dr. Th. Gabler Verlag, 1996. → Seite 2

[TS05] Can Türker und Gunter Saake. *Objektrelationale Datenbanken: Ein Lehrbuch.* dpunkt Verlag, 2005. → Seite 59, 66, 191

[Tür03] Can Türker. *SQL 1999 und SQL 2003: Objektrelationales SQL, SQLJ und SQL/XML.* dpunkt Verlag, 2003. → Seite 59

[UES03] Wolfgang Uhr, Werner Esswein und Eric Schoop. *Wirtschaftsinformatik 2003 Band I: Medien, Märkte, Mobilität.* Physica-Verlag, 2003. → Seite 13

[UML10] Object Management Group (OMG). *Unified Modeling Language (OMG UML), Infrastructure*, Mai 2010. Version 2.3. → Seite 110

[vGO26] Friedrich von Gottl-Ottlilienfeld. *Fordismus. Über Industrie und Technische Vernunft.* G. Fischer, Jena, 1926. → Seite 2

[Von11] Helmut Vonhoegen. *Einstieg in XML: Grundlagen, Praxis, Referenz.* Galileo Computing, 2011. → Seite 61

[W3C98] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0*, 1998. W3C Recommendation. → Seite 62

[W3C99] World Wide Web Consortium. *XML Path Language (XPath) Version 1.0*, 1999. W3C Recommendation. → Seite 61

[W3C01] World Wide Web Consortium. *XML Schema Part 0: Primer*, 2001. W3C Recommendation. → Seite 62

[W3C07] World Wide Web Consortium. *XQuery 1.0: An XML Query Language*, 2007. W3C Recommendation. → Seite 61

[Wes92] G. David A. Weston. National Language Architecture. In *CASCON '92: Proceedings of the 1992 Conference of the Centre for Advanced Studies on Collaborative Research*, Seiten 59–69, Toronto, Canada, November 1992. → Seite 41, 191

[WR05] Catharine M. Wyss und Edward L. Robertson. A formal characterization of PIVOT/UNPIVOT. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management (CIKM'05)*, Seiten 602–608, Bremen, Germany, November 2005. → Seite 92

[Wüp00] Josef Wüpping. Konfigurationstechnik für die individuelle Serienfertigung. *IT Management*, 7(4):2–9, 2000. → Seite 14, 15, 30

[ZHWG08] Kun Zhou, Qiming Hou, Rui Wang und Baining Guo. Real-time KD-tree Construction on Graphics Hardware. In *Proceedings of ACM SIGGRAPH Asia 2008*, Seiten 126:1–126:11, Singapore, 2008. → Seite 158

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Symbol-/Abkürzungsverzeichnis

$\perp$	Kennzeichen für einen unbelegten Aspekt (NULL-Wert) in Aspektsignaturen
$\ominus$	Kennzeichen für ein unbelegtes Attribut (NULL-Wert) in signierten Tupeln
$\leftrightarrow$	Kennzeichen einer bijektiven Abbildung $f : A \leftrightarrow B$
$\rightarrow$	Kennzeichen einer injektiven Abbildung $f : A \rightarrow B$
AF	Aspektfilter-Menge
Δ	Aspektabhängigkeitsfunktion anwendbar auf ein Relationenschema oder auf ein Attribut eines Relationenschemas
δ	Aspektabhängigkeitsfunktion zur Charakterisierung der Abhängigkeit von Attributen und Aspekten
$dom(A)$	Menge der Ausprägungen eines Aspekts A (Aspektschlüsselwertmenge)
$dom(C)$	Wertebereich eines Attributs C eines Relationenschemas
Σ	Menge aller Aspektsignaturen eines Relationenschemas
$\prod$	Kartesisches Produkt bei Anwendung auf Mengen
$r(R)$	Relation eines Relationenschemas R
$\wp(M)$	Potenzmenge von M, d.h. die Menge aller Teilmengen von M
A2C	Administration-to-Consumer
ACID	Atomicity/Consistency/Isolation/Durability, Transaktions-Eigenschaften
AOD	Aspektorientierte Datenhaltung
AOP	Aspektorientierte Programmierung
API	Application Programming Interface, Programmierschnittstelle
AST	Abstract Syntax Tree, Datenstruktur zur Syntaxprüfung
B2B	Business-to-Business
B2C	Business-to-Consumer

C2C Consumer-to-Consumer

CRM Customer Relationship Management, Verwaltung von Kundenbeziehungen

DBMS Datenbankmanagementsystem

DDL Data Definition Language, Befehle in SQL zur Schemadefinition

DML Data Manipulation Language, Befehle in SQL zur Datenmanipulation

EAI Enterprise Application Integration

EAV Entity-Attribute-Value, Datenmodell

ERP Enterprise Resource Planning, Steuerung von Unternehmensprozessen

EU Europäische Union, Verbund aus 28 Mitgliedsstaaten [Stand: 10.01.2014]

GUID Globally Unique Identifier, global eindeutige Zahl mit 128 Bit

GPU Graphics Processing Unit, Grafikprozessor

IAF Infix-Aspektfilter, siehe Definition 5.1

JDBC Java Database Connectivity, SQL-Schnittstelle für Datenbanken in Java

LDAP Lightweight Directory Access Protocol, Verwaltung von Zugriffsrechten

MVC Model-View-Controller, Architekturmuster

NF2 Non first normal form (NF^2), Aufhebung der ersten Normalform

NoSQL Not only SQL, Datenbanken mit nicht-relationalem Konzept

ODBC Open Database Connectivity, SQL-Schnittstelle für Datenbanken

OID Objektidentifikator, vom ORDBMS vergebene eindeutige Identifizierung

OLAP Online Analytical Processing

OOP Objektorientierte Programmierung

ORDBMS Objektrelationales Datenbankmanagementsystem

PDM Produktdatenmanagement, Verwaltung von Produktdaten

RDBMS Relationales Datenbankmanagementsystem

SGML Standard Generalized Markup Language, Meta-Auszeichnungssprache

SaaS Software as a Service

SQL Structured Query Language, Datenbanksprache für relationale Datenbanken

UML Unified Modeling Language, graphische Modellierungsprache für Software

VT Varianztest, Prüfkriterium beim Testdurchlauf

W3C World Wide Web Consortium, Standardisierungs-Gremium

XML Extensible Markup Language, erweiterbare Auszeichnungssprache

Selbstständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Hilfsmittel und Literatur angefertigt habe.

Jena, 10.01.2014

Lebenslauf

Persönliche Daten

Name	Matthias Liebisch
Geburtsdatum	2. März 1978
Geburtsort	Jena
Familienstand	verheiratet, 1 Kind

Schulische und universitäre Ausbildung

1984 – 1987	Polytechnische Oberschule „Wilhelm Pieck“ in Jena
1987 – 1991	Polytechnische Oberschule „Edwin Hoernle“ in Milda
1991 – 1996	Ernst-Abbe-Gymnasium in Jena, Abschluss mit Abitur
1997 – 2003	Informatik-Studium mit Nebenfach Mathematik an der Friedrich-Schiller-Universität Jena, Vertiefungsrichtung: Praktische Informatik, Schwerpunkt: Datenbanksysteme, Abschluss als Diplom-Informatiker

Zivildienst und beruflicher Werdegang

1996 – 1997	Zivildienst bei der Städtischen Fürsorge in Jena
seit 2003	Software-Entwickler/Projektleiter bei der ORISA Software GmbH
2007 – 2012	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Datenbanken und Informationssysteme der Fakultät für Mathematik und Informatik der Friedrich-Schiller-Universität Jena